NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

FRAMEWORK FOR EVALUATING LOOP INVARIANT DETECTION GAMES IN RELATION TO AUTOMATED DYNAMIC INVARIANT DETECTORS

by

Mehmet Yilmaz

September 2015

Thesis Advisor:	Geoffrey G. Xie
Second Reader:	Glenn Cook

Approved for public release; distribution is unlimited

REPORT DOCUMENTATION PAGE	*Form Approved OMB No. 0704–0188*

1. AGENCY USE ONLY ***(Leave blank)***	**2. REPORT DATE** September 2015	**3. REPORT TYPE AND DATES COVERED** Master's thesis

4. TITLE AND SUBTITLE FRAMEWORK FOR EVALUATING LOOP INVARIANT DETECTION GAMES IN RELATION TO AUTOMATED DYNAMIC INVARIANT DETECTORS	**5. FUNDING NUMBERS**
6. AUTHOR(S) Yilmaz, Mehmet	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000	**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A	**10. SPONSORING / MONITORING AGENCY REPORT NUMBER**

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government. IRB Protocol number ____N/A____.

12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited	**12b. DISTRIBUTION CODE**

13. ABSTRACT (maximum 200 words)

Software has a huge and negative impact on the economy. Formal verification is an effective method to check whether a piece of software contains certain kinds of errors. Defense Advanced Research Projects Agency (DARPA) started the Crowd Sourced Formal Verification (CSFV) program to propose a new model for formal verification by using five online games. CSFV aims to explore whether an online game player with no formal verification expertise can achieve formal verification more efficiently than through conventional processes. We observe that, currently, no quality criteria exist to measure CSFV gamers' efforts. The study suggests that machine detectability of a solution detected by a gamer indicates poor quality for that solution. The solutions in one of the games, StormBound, were selected for examination. An automated tool was developed to check the machine detectability of the solutions, and 78 percent of the assertions were seen to be machine detectable.

14. SUBJECT TERMS Crowdsourcing, software security, fuzzing, formal verification, automated loop invariant detection, loop invariant analysis, parser, Haskell language	**15. NUMBER OF PAGES** 85
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UU

THIS PAGE INTENTIONALLY LEFT BLANK

FRAMEWORK FOR EVALUATING LOOP INVARIANT DETECTION GAMES IN RELATION TO AUTOMATED DYNAMIC INVARIANT DETECTORS

Mehmet Yilmaz
Captain, Turkish Army
B.S., Turkish Military Academy, 2002

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN INFORMATION TECHNOLOGY MANAGEMENT

from the

NAVAL POSTGRADUATE SCHOOL
September 2015

Approved by: Geoffrey G. Xie
Thesis Advisor

Glenn Cook
Second Reader

Dan C. Boger, PhD
Chair, Department of Information Sciences

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Software has a huge and negative impact on the economy. Formal verification is an effective method to check whether a piece of software contains certain kinds of errors. Defense Advanced Research Projects Agency (DARPA) started the Crowd Sourced Formal Verification (CSFV) program to propose a new model for formal verification by using five online games. CSFV aims to explore whether an online game player with no formal verification expertise can achieve formal verification more efficiently than through conventional processes. We observe that, currently, no quality criteria exist to measure CSFV gamers' efforts. The study suggests that machine detectability of a solution detected by a gamer indicates poor quality for that solution. The solutions in one of the games, StormBound, were selected for examination. An automated tool was developed to check the machine detectability of the solutions, and 78 percent of the assertions were seen to be machine detectable.

TABLE OF CONTENTS

LIST OF FIGURES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF ACRONYMS AND ABBREVIATIONS

BOF	Buffer Overflow
CDF	Cumulative Distribution Function
CSFV	Crowdsourced Formal Verification
CVE	Common Vulnerabilities and Exposures
CWE	Common Weakness Enumeration
DARPA	Defense Advanced Research Projects Agency
LID	Loop Invariant Detection
MTurk	Mechanical Turk
NVD	National Vulnerability Database
OS	Operating System
SQL	Structured Query Language
VC	Verification Condition
VBA	Visual Basic for Applications

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

I would like to express my deepest appreciation to all those who provided me the opportunity to complete this thesis. I would like to express special gratitude to my advisor, Professor Geoffrey G. Xie from Computer Science Department, whose contribution in suggestions and encouragement helped me to write this thesis. In my life as a military officer and an academician, I will benefit from his unique style of generating the most precise question for a specific scientific problem.

I would also like to express my sincere gratitude to Professor Glenn Cook for serving as my second advisor and teaching me how to write a perfect research paper.

I would also like to thank to Umit Tellioglu, who aroused my curiosity about the Crowdsourced Formal Verification (CSFV); Andreas Baur, who accompanied me throughout the thesis project; and Charles Prince, who took great efforts to find solutions for the problems I encountered.

I would also like to thank to Aaron Tomb, who always gave detailed answers to my questions especially related to formal verification.

Last but not least, I would like to thank my family for supporting me throughout writing this thesis and my life in general.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

Software bugs continue to affect critical systems. Detecting them requires some mathematical models for understanding the behavior of a given software system. At present, the process of creating these mathematical models is time consuming and requires a high degree of mathematical and computer science training [1]. Worldwide, there are insufficient people with this training to verify every software system that would benefit from this approach [1]. The cost-effective formal verification will possibly increase the number of formally proved software [2].

The Defense Advanced Research Projects Agency (DARPA) published the Crowdsourced Formal Verification (CSFV) project in order to find a more affordable way of handling these challenges [3]. The goal of CSFV is to explore whether someone with no formal verification expertise can achieve formal verification more efficiently than through traditional methods [3]. With the DARPA CSFV program, a group of scientists developed a system to turn formal verification into five online games: Circuitbot, StormBound, Ghostmap, FlowJam, and one iOS game, Xylem—the code of plants. The games are published online at the Verigames.com website and were played by many gamers from various backgrounds.

The CSFV project aims to test the primary hypothesis of whether a non-expert Internet gamer can help verify a piece of software, which used to be the job of experts of computer programs. Computer programs are still unable to produce satisfactory results. Therefore, experts are still the best option for software verification. Even after the first phase of the CSFV project ended in September 2014, it is still unclear whether the contribution of CSFV gamers to software verification is satisfactory. The relative position of CSFV gamers to expert verifiers and computer verifiers is also unclear.

Two games were designed as loop invariant detecting (LID) games: StormBound and Xylem. They aim to present problems with the possible variable values from a particular loop. Gamers are expected to detect possible invariants (generalizable relations among the variables) among these values. Their answers (assertions) are written to the related loop as a comment that a software tester can use to see vulnerability. Similar to other formal verification methods, LID is the job of experts and computer programs (automated LIDs).

LID games represent one part of the CSFV project and await evaluation criteria just like other CSFV games. This thesis proposes quality criteria for LID games. An LID game needs to meet that criteria in order to have sufficient quality to make formal verification. This study proposes that the criteria be machine detectable. Machine detectable loop invariants refer to those that can be detected by an automated LID.

A. RESEARCH PROBLEM

LID games turn the invariant detection process into a problem-solving game. We observe that currently no criteria exist to measure the quality of gamers' efforts.

The primary purpose of this thesis is to propose criteria to measure the quality of LID gamers' effort. Those criteria will analyze the invariants detected by gamers of StormBound, one of the LID games of CSFV.

B. RESEARCH QUESTIONS AND HYPOTHESIS

This thesis tries to answer one primary question: How can we measure the quality of LID (StormBound) gamers' efforts?

One secondary question also needs to be addressed: What percentage of the invariant detected by LID (StormBound) gamers overlaps with the checklist of Daikon, an automated LID?

C. METHODOLOGY

This research focuses to determine whether StormBound gamers have detected some invariants that cannot be easily produced by Daikon. The thesis uses both qualitative and quantitative approaches. Auxiliary sources are studied primarily using a variety of materials. The StormBound developing team at Galois Company collected the data. This data consists of StormBound gamers' invariants, which were written in Haskell Language.

By examining this data and the manual of Daikon (an automated LID), we recognized that some of the gamers' invariants are included by a Daikon checklist (see Appendix.B). Daikon's variants are known to have poor quality. We suggest that StormBound loop invariants overlapping with the checklist of Daikon are poor quality.

The data we received on May 18, 2015, contains 48,244 assertions produced by StormBound gamers. Because it is hard to analyze that much data, we developed a tool that checks each of these assertions for detectability by Daikon. The checker is an Excel Visual Basic for Applications (VBA) script that checks the overlapping invariants in a given list by comparing them under the assumption defined by the user.

D. POTENTIAL BENEFITS

Firstly, game developers can use the criteria to quantify the quality of gamers' efforts.

Secondly, project managers can use the criteria as a parameter to determine the return of investment of the CSFV project.

E. SCOPE AND LIMITATIONS

In this thesis, we used the raw data received from one VeriGames' developer related to the StormBound game. However, we do not have raw data to examine the other game Xylem, which has the similar mechanism.

Secondly, we used abstraction to simplify the highly complex data, such as translating a first-level operation (addition, subtraction, etc.) for two variables into another variable. That approach is likely to decrease the accuracy of the finding.

Thirdly, we used only Daikon as an automated LID. However, there are many other tools for similar purposes. This study covers only the Daikon checklist.

F. THESIS ORGANIZATION

In Chapter I, we introduced the CSFV project by elaborating its main intent of creating a new model for checking the software bugs with crowdsourcing. We also explained the problem and purpose of this thesis.

In Chapter II, we discuss the concepts used in this thesis. We explain software security and its importance in today’s highly technological world. Then we explain the two major techniques used for software security: testing and formal verification. We also explain loop invariant analysis, automated LID, crowdsourcing, the CSFV project, StormBound and Xylem, and the Haskell Language.

In Chapter III, we explain how we got the data. Then we introduce the method we used to read it. The data is too complex for a non-expert to understand, so we introduce the abstraction method that we used to make the data more understandable.

In Chapter IV, we introduce why we need such criteria. We introduce the mechanism of automated LIDs. Finally, we introduce our checker tool developed for detecting whether a given loop invariant is detectable by Daikon.

In Chapter V, we reveal our findings. We also explain the limitations of the study and what can be done in further studies.

II. BACKGROUND

This chapter covers the basic concepts that will help readers understand the thesis. Section A details software security and software vulnerabilities because the CSFV project aims to find a more affordable way to secure software free of certain types of vulnerability. Section B elaborates on the two major methods of finding security vulnerabilities: testing and formal verification. Formal verification is the most effective way to ensure that software is free of certain types of vulnerability. Formal verification can be performed with many methods, including loop invariant analysis. Section C elaborates conventional loop invariant detection processes. The project aims to attract online gamers who can detect loop invariants in a piece of software. Some computer programs can also be used for the same purpose. Section D introduces Daikon as a computer-based loop invariant analysis tool. It is a machine learning tool that detects invariants in a given piece of code. Section E mentions the CSFV project while explaining why DARPA performs its research and how the CSFV system works. The CSFV project mainly depends on online games developed by various companies for that special project. This study examines StormBound which is one of the five CSFV games. Section F details that game.

A. SOFTWARE SECURITY AND BUGS

A bug or vulnerability in the OpenSSL software is defined as a problem in any software. In April 2014, the release of Heartbleed vulnerability created a big concern for the security of the Internet. One of the most significant Internet weaknesses, Heartbleed enables hackers to distantly read memory information from various popular HTTPS sites including many commercial one such as Amazon, EBay [4]. The release of similar vulnerabilities negatively affects general attitude towards Internet. People are less likely to trust the Internet if they are worried that they will lose their personal information and their money.

With the increased Internet usage, regular Internet users' concern for Internet security has increased [5]. Critical Internet services are supposed to be free from security vulnerabilities. Besides Internet services, other critical systems such as Avionics can be disastrously affected by similar weaknesses. Software security vulnerability in the electronic systems of an aircraft can cause many people to lose life [6]. A range of consequences of software errors have been reported as shocking news, reducing people's trust in software. Some of these incidents are related to English pensioners incorrectly identified dead, an innocent man who was almost imprisoned [7].

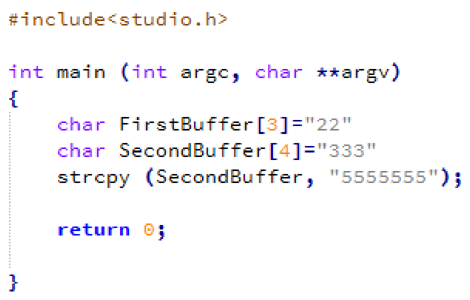
Information technologies are in every part of our daily lives. We are surrounded by many software systems: computers, smartphones, and tablets. All these vital technologies have two main components: software and hardware. Besides their benefits, they may also be harmful. On the one hand, we use smartphones to share pictures with our family; on the other hand, hackers may access our accounts and steal our credit card numbers. All software users, from smartphone users to an airplane pilots, are aware that software security is vital for the safe and reliable performance of any given device. With the increase in commercial and financial opportunities online, security concerns also increase. Cyberspace does not have boundaries. As human beings, we can choose our friends. Online, hackers are no further than our closest friends. Under these pessimistic conditions, many people choose not to become involved in or over-reliant on technology. Fortunately, experts are working to make these devices free from security concerns.

Today's software is intricate and available to almost any user. It is developed in diverse languages and performs on diverse execution environments, such as browsers, language processors, and databases [8]. Software has millions of lines of code, as well as different functions and components from very different platforms. Just as any other human-produced technology, it may include some failure points inside that complex structure. Some part of that huge software is likely to fail. Failure can cause a temporary cessation of services stop or result in credit card theft. Although a temporary cessation of services seems trivial, its timing defines how critical it may be. In the case of an emergency situation, a temporary cessation of services may result in loss of life.

Some researchers are looking for the possible failures in software by using either formal methods or testing. A variety of tools and methods are used to develop and test the software before actual utilization to guarantee that specific requirements are met [8]. In a different study, systematic test of the effectiveness of programs running on various versions of the UNIX operating system was conducted [9].

Even if there are many testing tools and methods, they are still inadequate to prevent discovery of new vulnerabilities [10]. Hackers can use vulnerabilities to steal data from a critical system. In order to explain vulnerability, we present one example. One of the most commonly known vulnerabilities is buffer overflow (BOF) vulnerability. BOF vulnerability is a weakness that enables a code to exceed the allocated size for data to overwrite to a different location in memory. According to [11], "The overwriting corrupts sensitive neighboring variables of the buffer such as the return address of a function or the stack frame pointer."

Figure 1. A Simple Code for String Operations



```
#include<studio.h>

int main (int argc, char **argv)
{
    char FirstBuffer[3]="22"
    char SecondBuffer[4]="333"
    strcpy (SecondBuffer, "5555555");

    return 0;

}
```

The C code in Figure 1 is a simple example for BOFs. The code assigns two variables with the size of three and four. The strcpy function copies seven characters into the address place of "SecondBuffer" variable. The "SecondBuffer" variable is allocated enough space to hold four characters in the memory and three characters and a NULL value (see Figure 2). When the strcpy function runs, the first three characters of the "5555555" are written on the "SecondBuffer," and first two of the rest are written on the "FirstBuffer."

Figure 2. Memory before Strcpy Function

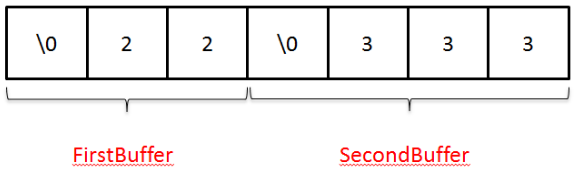
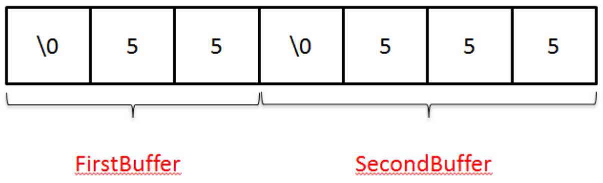


Figure 3 shows the memory after the strcpy function execution. “FirstBuffer” value was changed undesirably to “55” from “22.” This is a simple example for BOF vulnerability. Some part of the code made an undesired change on the memory. If the “FirstBuffer” variable has a critical function on the overall application, then after the execution of strcpy this function may crash. Considering the possibility of that function being a critical banking service, that crash may create high level financial consequences.

Figure 3. Memory after Strcpy Function



There are many other vulnerabilities similar to BOF, such as SQL injection, command injection, cross-site scripting, and missing encryption. Every day, we encounter news a financial damage caused by a new software security vulnerability [12]. There are some online repositories used to publish software vulnerabilities and their solutions, such as the National Vulnerability Database (NVD) [13], Common Vulnerabilities and Exposures (CVE) database [10]).

B. TESTING AND FORMAL VERIFICATION

A vulnerability or bug occurs when the software does something that it is not supposed to do. In order to detect such errors and develop software free of certain types of vulnerability for critical functions, it is important to understand the characteristics of software testing. The primary purpose of any software testing method is to monitor the reactions of software in the production environment and to detect the unexpected reactions. We test a system or its components in order to be sure that it satisfies the defined requirements. In other words, we want to guarantee that the reactions of a system are desirable.

According to [14], “Although there has been a significant increase in security awareness among software developers during the past few years, there are still many developers who do not have the necessary expertise in developing secure programs.” The shortage in expertise on the developers made the security expert to develop automated testing tools such as Fuzz testing tools. Professor Barton Miller and his students from University of Wisconsin Madison have invented the first fuzz testing tool in 1989 [15]. Fuzzing is a computerized testing which is conducted to trigger a crash in the application in order to disclose the mistakes inside of it [16]. Fuzzing is a very popular way of testing.

Verifying software guarantees that a specific portion of software is not affected by particular mistakes [2]. Since mainly expert engineers are needed to perform software verification, verifying large software systems requires big budgets [2]. Formal verification is becoming a more critical element with the increase of complex applications. While hardware complexity increases parallel to Moore’s Law, verification complexity is growing even more with a speed twice that of hardware complexity [17]. Since verification phase takes almost 70% of the development time, verification complexity is considered the main bottleneck for design [17].

Formal verification includes mathematical reasoning to confirm that a design specification is met during the execution of the system [17]. Formal verification is used to detect the vulnerabilities, such as the BOF inside the code. Loop invariant analysis is a method used for formal verification.

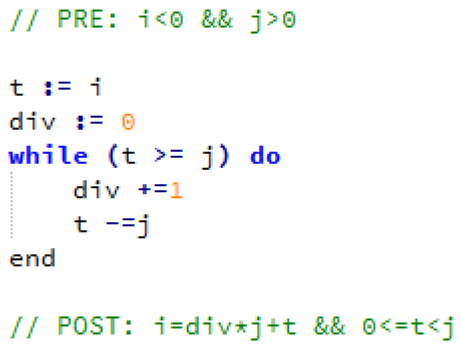
C. LOOP INVARIANT ANALYSIS

In computer science, a *loop* is a control statement programming language for identifying iteration, which allows code to be executed repetitively. Due to the fluctuating organization of loops, loops are more likely to have a bug than the other parts of the code.

It is hard to reason the behavior of a loop because of the ambiguity in the number of iterations. Furthermore, the iteration number may be subject to the input data, which makes determining all possible combinations too hard. According to [18], “One solution to this problem is to reason about loops independently of the number of iterations: loop invariants are logical statements that describe properties of a loop holding for all possible executions of the loop.” Loop invariants are the values indicating the relationship between a single variable and a value or relationship between many data members [19].

According to [20] “An invariant is a property that holds at a certain point or points in a program; these are often used in assert statements, documentation, and formal specifications. Examples include being constant (x = a), non-zero (x 6= 0), being in a range (a ≤ x ≤ b), linear relationships (y = ax + b), ordering (x ≤ y), functions from a library (x = fn(y)), containment (x ∈ y), sortedness (x is sorted), and many more.” A *loop invariant* is a scheme composed of variables from the program that are true prior to the loop, through iterations of the loop, and after the loop [21]. For example, in Figure 4, one can show that the statements (t >=0) and (i = div * j + t) are invariants for the loop [21].

Figure 4. A Simple Loop Program



```
// PRE: i<0 && j>0

t := i
div := 0
while (t >= j) do
    div +=1
    t -=j
end

// POST: i=div*j+t && 0<=t<j
```

Testers benefit from invariants in order to better understand an application [20]. Loop invariants describe the behavior of a loop. The behavior should be the same in all possible iterations of the loop and can be used to understand the existence of a bug inside the loop. The number 2,147,483,647 is the maximum positive value for a 32-bit signed binary integer in computing. Considering a variable “x” as integer in a loop and a loop invariant that shows that x can get a value greater than the maximum positive value for a 32-bit signed binary integer, we see a problem. When the x gets a greater value than it can store in the allocated space in memory, it writes the part that does not fit into its allocated space into some other location that may be used by another variable. The unintentionally changed variable may stop the code.

Loop invariants are similarly useful for testing. Testers can use them to generate better test cases [22]. They can change the program verification process dramatically and drastically speed up processes like automatic test case generation [18]. Having recognized the loop invariants in the initial phases of software development, programmers can detect the properties that produce the correct requirements [23].

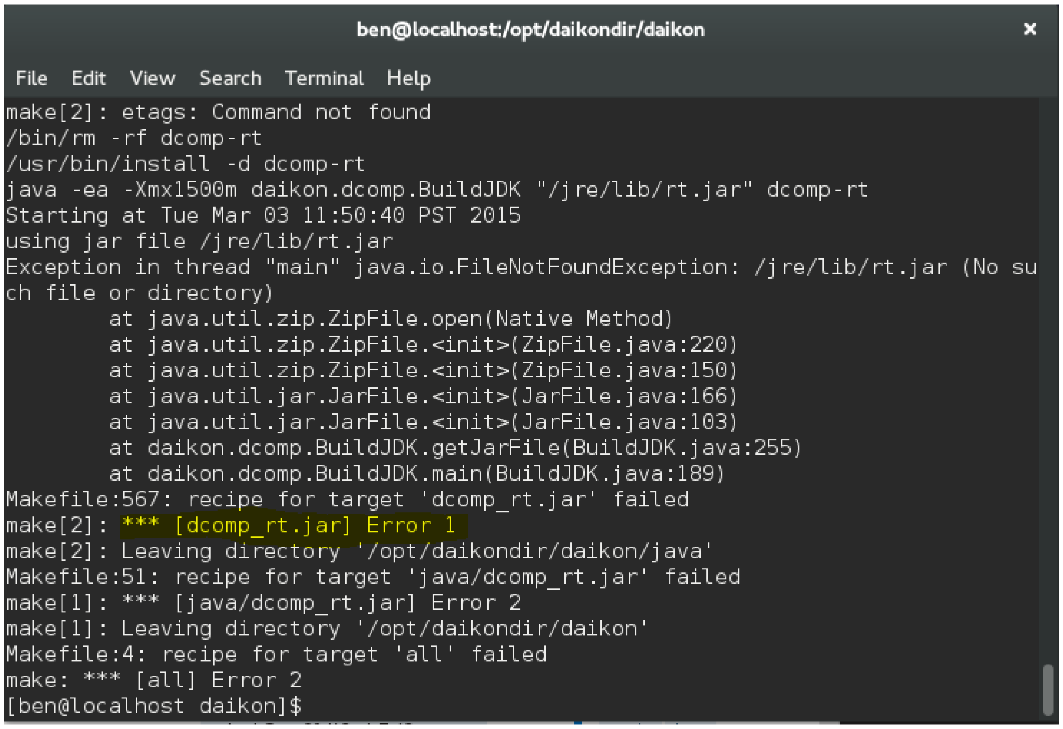
D. AUTOMATED INVARIANT DISCOVERY SYSTEMS (DAIKON)

In the previous example, the loop invariants may seem easy to detect. When it comes to the more complex applications with millions of lines of code, detecting loop invariants is not that simple. Generally speaking, only highly educated computer scientists can perform such a complex task. Considering the cost of formal verification by experts, some scientists study computer programs for formal verification.

According to [19], “at present, most invariant generation is performed manually, and tools for automating invariant detection are limited in power and effectiveness. First, invariant detectors cannot discover a complete set of invariants, because the problem of determining all invariants is undecidable. Second, invariant generation tools usually operate at the level of functional granularity, partly because the idea for invariants developed from precondition, post condition and loop invariants, which are derived from methods that consider basic blocks, and partly because statement analysis is extremely expensive in terms of time.”

In 2006, Ernst et al. introduced “the Daikon system for dynamic detection of likely invariants, which is an implementation of dynamic detection of likely invariants; that is, the Daikon invariant detector reports likely program invariants” [20]. These invariants are written before and after the related function to be used by the developer to detect any unintended reaction. Dynamic invariant detection executes a piece of code, detects the values that it computes, and then reports the invariant (relation), which is true over the detected executions [20]. Moreover, it is a machine learning technique for arbitrary data to detect invariants in C, C++, Java, and Perl programs, and in record-structured data sources; Daikon can easily be extended to other applications [20].

Figure 5. Installation of Daikon


ben@localhost:/opt/daikondir/daikon
File Edit View Search Terminal Help

```
make[2]: etags: Command not found
/bin/rm -rf dcomp-rt
/usr/bin/install -d dcomp-rt
java -ea -Xmx1500m daikon.dcomp.BuildJDK "/jre/lib/rt.jar" dcomp-rt
Starting at Tue Mar 03 11:50:40 PST 2015
using jar file /jre/lib/rt.jar
Exception in thread "main" java.io.FileNotFoundException: /jre/lib/rt.jar (No su
ch file or directory)
        at java.util.zip.ZipFile.open(Native Method)
        at java.util.zip.ZipFile.<init>(ZipFile.java:220)
        at java.util.zip.ZipFile.<init>(ZipFile.java:150)
        at java.util.jar.JarFile.<init>(JarFile.java:166)
        at java.util.jar.JarFile.<init>(JarFile.java:103)
        at daikon.dcomp.BuildJDK.getJarFile(BuildJDK.java:255)
        at daikon.dcomp.BuildJDK.main(BuildJDK.java:189)
Makefile:567: recipe for target 'dcomp_rt.jar' failed
make[2]: *** [dcomp_rt.jar] Error 1
make[2]: Leaving directory '/opt/daikondir/daikon/java'
Makefile:51: recipe for target 'java/dcomp_rt.jar' failed
make[1]: *** [java/dcomp_rt.jar] Error 2
make[1]: Leaving directory '/opt/daikondir/daikon'
Makefile:4: recipe for target 'all' failed
make: *** [all] Error 2
[ben@localhost daikon]$
```

Daikon is a tool for dynamically detecting potential invariants and it can be downloaded from a download site with unrestricted use [24]. The Figure 5 shows one part of the installation process of installing Daikon.

There are some problems for the existing dynamic invariant detection method. Firstly, Daikon can only analyze some of the variables: parameters and return values [19]. Secondly, its checklist is extremely limited [19]. It can only detect invariants with one, two, and three variable.

E. CROWDSOURCED FORMAL VERIFICATION

Crowdsourcing means using a large number of Internet users to perform a particular job. In Brabham's 2008 article [25], crowdsourcing was viewed as an online model for distributed production and problem-solving. Since 2006, many examples emerged in different fields. Amazon Mechanical Turk (MTurk) is a well-known example of crowdsourcing.

MTurk is a crowdsourcing Internet platform on which users can find other people to perform some task that is impossible to do by automation. It is an online website for researchers to post annotation tasks so that they can be done by regular users for a small payment [26].

Foldit is another example of crowdsourcing on the gaming industry. Foldit is an online game that enables gamers to create correct protein structure models [27]. Foldit showed that the potential gamers are likely to solve harder academic problems [28].

Ernst et al. [2] studied ways to leverage formal verification by using game-playing-based verification systems. They introduced the special online game "Pipe Jam," which turns formal verification into a fun game for Internet users. According to [2], "to remap the problem into a more accessible form, and use an engaging game to develop a significantly larger number of experts capable of solving verification problems in a remapped domain. Instead of relying on software engineers, we will develop a new skilled verification workforce, and use crowd-sourcing on a much more general audience of people who enjoy the challenge of playing a game."

Crowdsourcing is a paradigm shift for many tasks, from coding to surveying. Many companies are benefiting from that new approach. The CSFV program was developed by DARPA to overcome the challenges related to the fact that the formal verification techniques are expensive and time consuming [3]. Its main object is to examine whether formal verification can be done by a large number of non-experts faster and more cost-effectively than by current approaches [3]. Turning software verification into a game for Internet users is the main goal of the project [3]. According to [3], "The program envisions numerous benefits, including: increased frequency and cost-effectiveness of formal verification for more types of common COTS software; greatly expanded audience to participate in formal verification; establishment of a permanent community of game players interested in improving software security." The CSFV project basically aims to benefit from the efforts of online game by turning their efforts into mathematical proofs that can be used to formally verify particular software. This project may cause one regular gamer to find a bug similar to Heartbleed.

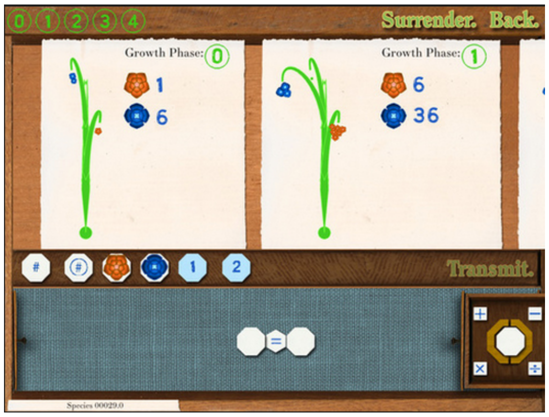
There are five games hosted on the website Verigames [3]:

- CircuitBot: "Link up a team of robots to carry out a mission."
- Flow Jam: "Analyze and adjust a cable network to maximize its flow."
- Ghost Map: "Free your mind by finding a path through a brain network."
- StormBound: "Unweave the windstorm into patterns of streaming symbols."
- Xylem: "Catalog species of plants using mathematical formulas."

StormBound and Xylem are two games specifically focused on loop invariant detection. This study mainly focuses on StormBound. However, we need to give some general information concerning Xylem to make the games related to LID. The goal of Xylem is similar to StormBound. Both aim to make non-expert gamers contribute to LID. It is an iPad game where gamers make mathematical observations about synthetic plants, and thereby contribute to the formal modeling of a software system [1]. Specifically, flowers found on the plants represent the value of variables found inside a source code loop (for, while, do), and gamers are asked to find relationships that describe the number of flowers [1].

Figure 6 shows a simple chapter of the game. The numbers next to the flowers are the values of two different variables in a loop. The game requires the gamer to create an equation from these numbers. For that special example, one solution might be (Orange flower) x 6 = (Blue flower).

Figure 6. Xylem Game Interface



If the mathematical model is an accurate enough representation of the functionality of the software system, then the outputs of the automated reasoning tools are accurate statements about the software system itself [1]. The classic example of a bottom-up, data-driven approach is Daikon [20], which uses analyst-provided templates to synthesize possible invariants among the variables of interest, discarding the propositions that are not attested by concrete test cases. One particular use case where bottom-up approaches excel—and the case for which Xylem is intended—is in annotating large legacy systems with loop invariants to prove broad categories of safety and security properties [1].

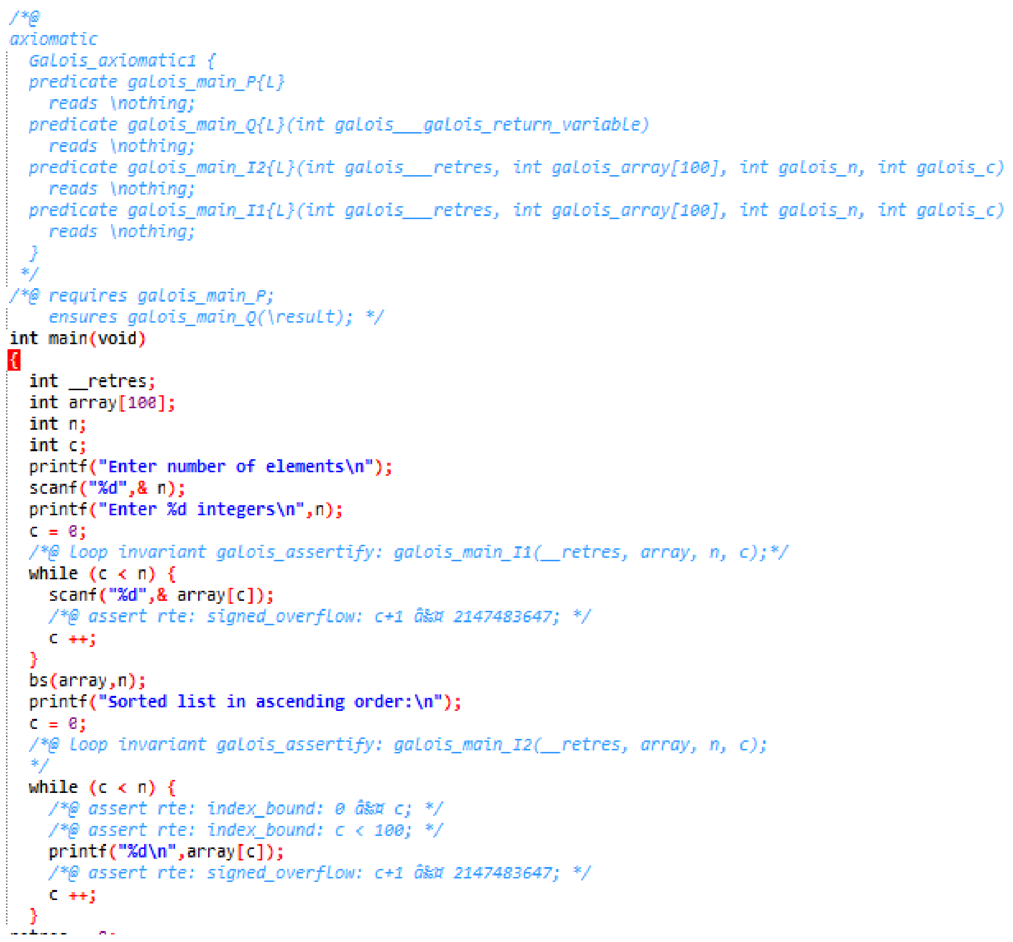
F. STORMBOUND

StormBound was developed by scientists from Galois Company, specialists in formal methods, and VoidALPHA Company. They developed StormBound in the first phase of the project. In a personal interview, Aaron Tomb explained the logic behind the StormBound game as follows: Formal verification is a mathematical approach to software security. StormBound was developed to contribute to formal verification by using the math skills of the gamers.

Frama-C stands for Framework for Modular Analysis of C programs. It is a set of interoperable program analyzers for C programs, which enables the analysis of C programs without executing them. Frama-C gathers several static analysis techniques in a single collaborative framework [29].

The normal workflow of Frama-C is simple. First, verification experts annotate programs with invariants into Frama-C. Then, Frama-C attempts to prove that those invariants are actually valid for the underlying program by automatically generating a collection of verification conditions (VCs). These VCs are purely logical formulas which, if they can be proved, show that the invariants are true invariants of the program. Figure 7 shows a code that was annotated by Frama-C.

Figure 7. A Part of Code Annotated by Frama-C

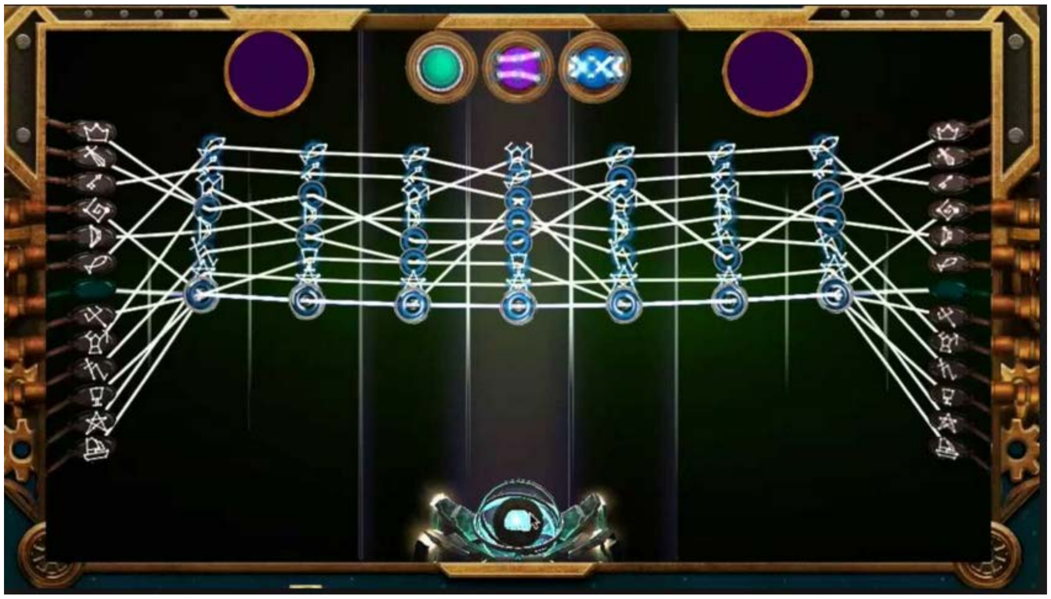


```
/*@
axiomatic
  Galois_axiomatic1 {
  predicate galois_main_P{L}
    reads \nothing;
  predicate galois_main_Q{L}(int galois___galois_return_variable)
    reads \nothing;
  predicate galois_main_I2{L}(int galois___retres, int galois_array[100], int galois_n, int galois_c)
    reads \nothing;
  predicate galois_main_I1{L}(int galois___retres, int galois_array[100], int galois_n, int galois_c)
    reads \nothing;
  }
 */
/*@ requires galois_main_P;
    ensures galois_main_Q(\result); */
int main(void)
{
  int __retres;
  int array[100];
  int n;
  int c;
  printf("Enter number of elements\n");
  scanf("%d",& n);
  printf("Enter %d integers\n",n);
  c = 0;
  /*@ loop invariant galois_assertify: galois_main_I1(__retres, array, n, c);*/
  while (c < n) {
    scanf("%d",& array[c]);
    /*@ assert rte: signed_overflow: c+1 â‰¤ 2147483647; */
    c ++;
  }
  bs(array,n);
  printf("Sorted list in ascending order:\n");
  c = 0;
  /*@ loop invariant galois_assertify: galois_main_I2(__retres, array, n, c);
  */
  while (c < n) {
    /*@ assert rte: index_bound: 0 â‰¤ c; */
    /*@ assert rte: index_bound: c < 100; */
    printf("%d\n",array[c]);
    /*@ assert rte: signed_overflow: c+1 â‰¤ 2147483647; */
    c ++;
  }
```

StormBound verification infrastructure made small changes on that flow. Instead of having experts insert annotations describing invariants, the verification infrastructure inserts placeholders. It assumes that the given loop should have a loop invariant, so it produces a name such as I1 and creates an abstract predicate with that name, parameterized by all of the program variables in scope. Considering the program variables x, y, and z, it would add an annotation such as (/*@ loop invariant I1(x, y, z); */) before the loop. Once annotations exist in all the appropriate places, Frama-C can generate verification conditions. At this point, we do not know the predicate for I1. It can be “x < y,” “x + y = z - 1,” or anything else mentioning those variables. The blue printed material in the figure-x shows the annotations added by Frama-C. They just have the name of the variables.

Thus, there are no concrete invariants, only automatically-added abstract predicates. The gamer’s job in StormBound is to figure out what all uninterpreted predicates should be. In other words, gamers are expected to turn the (/*@ loop invariant I1(x, y, z); */) into something like “x < y,” “x + y = z - 1.” The key problem is that some of the required invariants are complex. Therefore, it could be that game gamers have a hard time discovering the complex invariants. However, because abstract predicates are introduced for every relevant location, human experts could fill them in later, as if they were gamers, but by directly specifying them instead of playing the game. Figure 8 shows an example of the StormBound game phases.

Figure 8. StormBound Game Interface



In other words, a gamer sees the overall proof structure or part of it while trying to generate new assertions. Many of vulnerabilities exist because of undefined behavior in the C language. Frama-C has a built-in notion of what is fully-defined by the C standard and includes in its VCs checks that the program is operating within the well-defined subset of the language. The Frama-C work has established that certain Common Weakness Enumerations (CWE) can never happen when the code follows a well-defined subset of the C language, and the VCs mainly confirm no violation of that subset.

The only manual effort from people other than gamers required in our current game is in these areas:

- Identifying the particular bits of source code to analyze and feeding them into the system.
- Putting in any additional annotations about high-level properties to check (unnecessary if you just want to check for memory safety/well-defined behavior).

In case of an abstract predicate for two variables, there are many possible subsets of x and y. The StormBound brings some of them to gamers so that they can see a general relationship between them such as (x>y). The main difficulty is that, in practice, there are close to an infinite number of possible values, so it is impossible to test them all individually. In StormBound, seeing a representative sample of possible values would be enough to help gamers discover relevant patterns.

G. SUMMARY

Crowdsourcing is a relatively novel approach. Many new business ideas are using that approach in their business model. Amazon MTurk is the most commonly known. Crowdsourcing is still continuing to attract new ideas. This evolution merged with the problems of software security in the CSFV project. CSFV is a new model for detecting software bugs with crowdsourcing. CSFV scientists developed five online games. One of them, StormBound, expects its gamers to detect the possible loop invariants in a given piece of software. The gamers are not aware of the related piece of software. They are attempting to solve the problems similar to other puzzle games.

THIS PAGE INTENTIONALLY LEFT BLANK

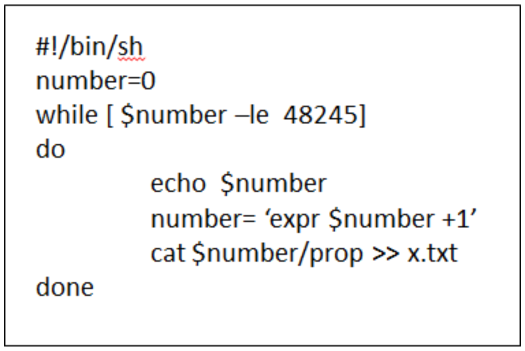
III. DATA ABSTRACTION

This chapter covers the content of the data. The data provided by StormBound developers requires getting the data from files scattered among hundreds of different subfolders. Beside that complex location system, the data consists of many assertions that are a combination of many interrelated functions. Section A explains the complexity of the data. Section B proposes a data translation method that makes the data easy to understand. And Section C explains how we manage the data by using a parser that we developed for that particular problem.

A. DATA COMPLEXITY

For this research, no new data has been collected. Galois Company directly provided the data that they have collected from December 2013 to May 2015. Therefore, we did not have a control over which types of data were collected. The dataset includes 48,244 assertions generated by from StormBound gameplay and is considered valid by the StormBound backend. The data provided by StormBound developers consists of 4,473 distinct folders and subfolders. The assertions were stored inside "prop" files in the least level folders. It is necessary to pull the data from these "prop" files and push than into a single file.

Figure 9. Shell Script to Pull Data from Files



```
#!/bin/sh
number=0
while [ $number –le  48245]
do
            echo  $number
            number= 'expr $number +1'
            cat $number/prop >> x.txt
done
```

The folders are named after connected functions, such as "acache_cancelentry" or "tostruct_in_dhcid." Inside each of these folders are other folders with the name ending with P (), Q(), and I1(). *P* refers to the pre-condition, *Q* refers to post-condition, and *I* refers to the loop invariants. Inside some of these subfolders were folders named “human.” That indicates that gamers created an invariant for that part of the function. Inside that are the numbered folders, all of which have a single file named “Prop.” Assertions were stored inside these files. Using the shell script (see Figure 9), we copies 48,244 assertions from the “prop” files into a single text file.

Figure 10 shows three examples of these assertions, which are written in Haskell programming language, a standardized and general-purpose purely functional programming language [30]. Assertions are combinations of many logical and mathematical functions. Each additional function in an assertion increases its complexity.

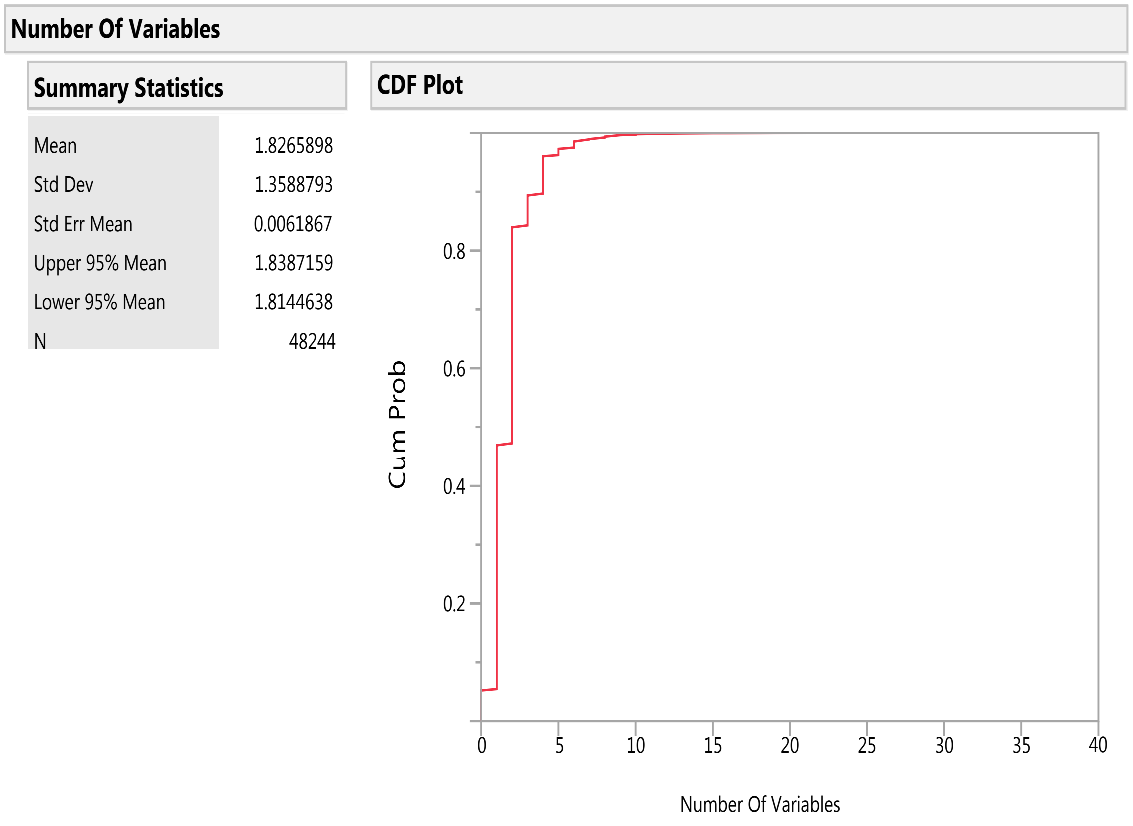
Figure 10. Examples of the Assertions

FOr (FOr (FOr (FOr (FOr (FOr (FOr (FOr (FEq (Sub (Const) (GetField (Deref (Var "m")) "free_query")) (Add (Mul (Const 1) (Const (-1))) (Const 1))) (FEq (Sub (Const) (GetField (Deref (Var "m")) "free_query")) (GetField (Deref (Var "m")) "flags"))) (FEq (Sub (Const) (GetField (Deref (Var "m")) "free_query")) (GetField (Deref (Var "m")) "free_query"))) (FEq (Sub (Const) (GetField (Deref (Var "m")) "free_query")) (GetField (Deref (Var "m")) "free_saved"))) (FEq (Sub (Const) (GetField (Deref (Var "m")) "free_query")) (Const))) (FEq (Sub (Const) (GetField (Deref (Var "m")) "free_query")) (GetField (Deref (GetField (Deref (Var "m")) "cctx")) "magic"))) (FEq (Sub (Const) (GetField (Deref (Var "m")) "free_query")) (GetField (Deref (GetField (Deref (Var "m")) "cctx")) "allowed"))) (FValid (GetField (Deref (Var "m")) "cctx") (Const) (Const))) (FNot (FValid (GetField (Deref (Var "m")) "cctx") (Const) (Const)))
FGt (GetField (ArrayIx (Var "task") (Const)) "impmagic") (GetField (ArrayIx (Var "task") (Const)) "magic")
FGt (GetField (Deref (Var "dctx")) "allowed") (GetField (Deref (Var "dctx")) "magic")

In Haskell, the function names are written before the parameters. For example, `FEq(Var “x”)(Var “y”)` indicates that the variable “x” is equal to variable “y.” For readers, it is exceptionally hard to understand the assertions with more than three functions. Therefore, a translation of the data from the current format to a more comprehensible format is crucial to analyze it.

Figure 11 shows the cumulative distribution function diagram of the number of variables in each StormBound loop invariant. Almost 80 percent of Stormbound loop invariants has less than three variables. That shows that the data is not too complex in terms of the variables. Secondly, the average number of variables is 1.8. This study does not differentiate the variables with the same name. All variables indicate a different variable than the other variables in the same loop invariant even if their names are same. That is a limitation of the study. More accurate analysis is assumed to refer to a lesser complexity level.

Figure 11. Variables CDF Diagram



Number Of Variables

Summary Statistics

Mean	1.8265898
Std Dev	1.3588793
Std Err Mean	0.0061867
Upper 95% Mean	1.8387159
Lower 95% Mean	1.8144638
N	48244

Figure 12 shows the cumulative distribution function diagram of the number of parameters in each StormBound loop invariant. Parameters refer to the expression names such as Add, Sub, Mul, Div, Mod, Deref, SizeOf, ArrayIx , FNot, FOr, FEq, FGt, FNeq, FValid. Almost 80 percent of Stormbound loop invariants has less than three variables. That shows that the data has moderately complex in terms of the variables. Secondly, the average number of variables is 2.9. This study does not differentiate the parameters according to their contribution to the complexity of a loop invariant. Parameters such as FEq an FGt adds more complexity than parameters such as Add, Sub. That is a limitation of the study. More accurate analysis is assumed to refer to a higher complexity level.

Figure 12. Parameters CDF Diagram

Number Of Parameters

Summary Statistics

Mean	2.9417751
Std Dev	3.0510632
Std Err Mean	0.0138909
Upper 95% Mean	2.9690014
Lower 95% Mean	2.9145488
N	48244

CDF Plot

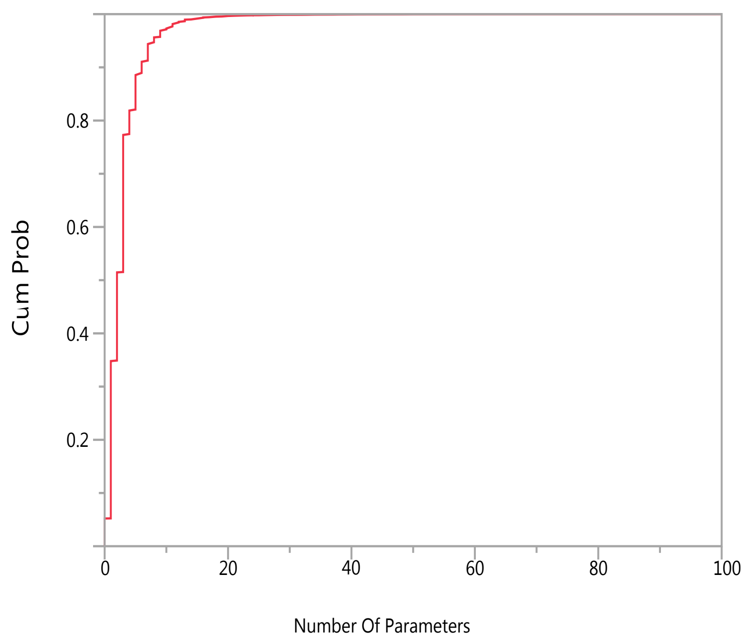
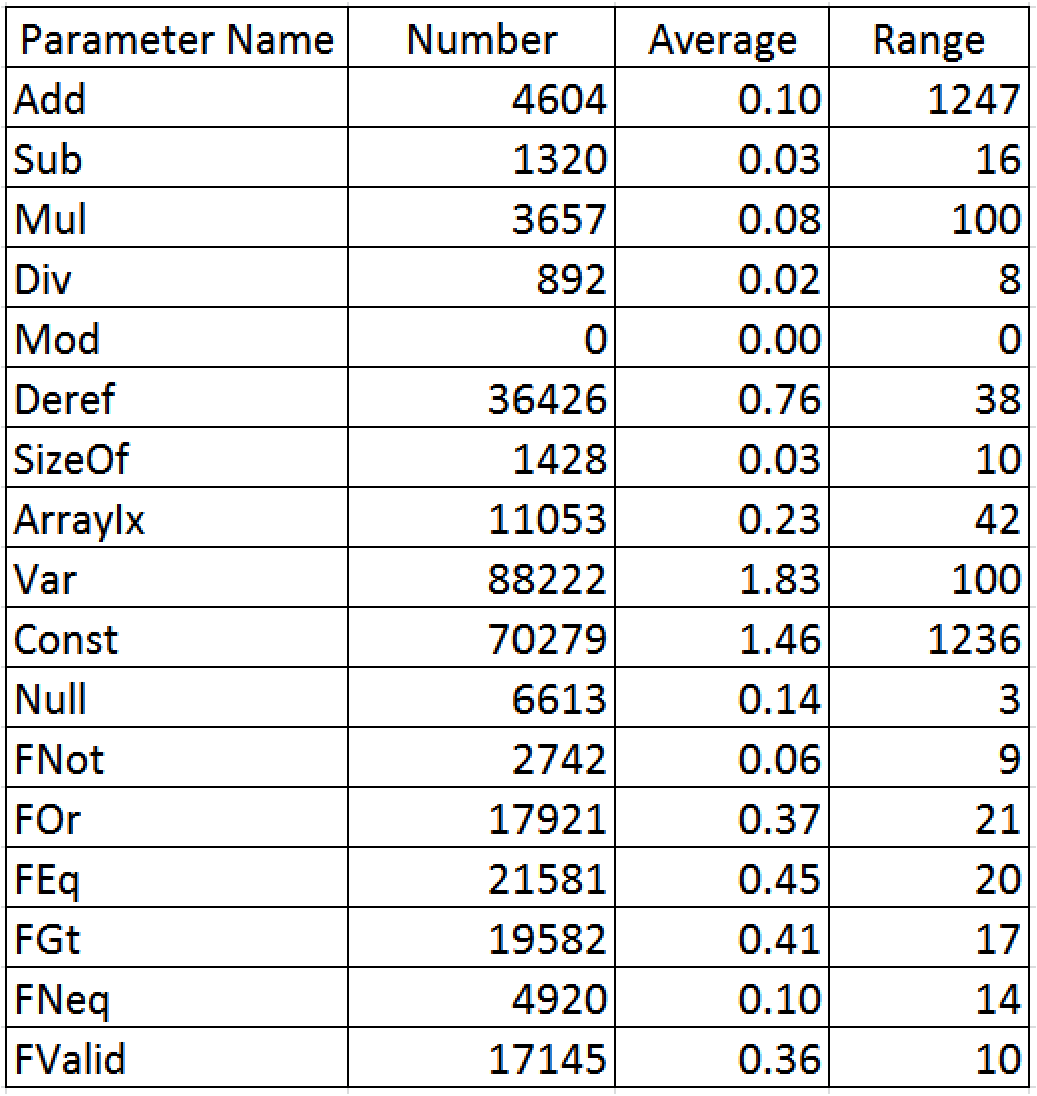


Figure 13 shows the statistics about the parameters of StormBound loop invariants in detail. It shows how many times a particular parameter used in total, what the average number of usage of any particular parameter in each loop invariants is what the range of any particular parameter is. This table shows that some of the parameters were used more than the others. As an example, the number of Deref is 36426, while the number of SizeOf is 1428.

Table 1. Statistical Information about StormBound Loop Invariants



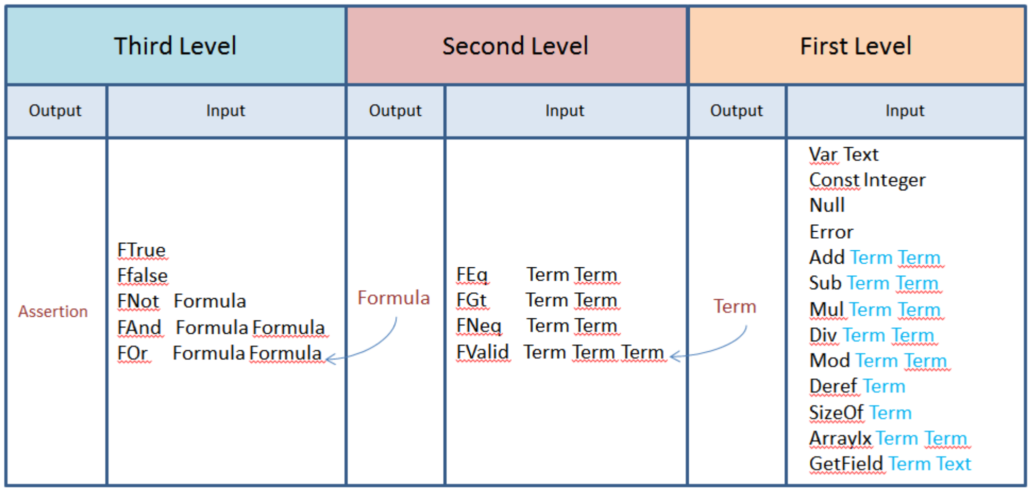
Parameter Name	Number	Average	Range
Add	4604	0.10	1247
Sub	1320	0.03	16
Mul	3657	0.08	100
Div	892	0.02	8
Mod	0	0.00	0
Deref	36426	0.76	38
SizeOf	1428	0.03	10
ArrayIx	11053	0.23	42
Var	88222	1.83	100
Const	70279	1.46	1236
Null	6613	0.14	3
FNot	2742	0.06	9
FOr	17921	0.37	21
FEq	21581	0.45	20
FGt	19582	0.41	17
FNeq	4920	0.10	14
FValid	17145	0.36	10

B. DATA TRANSLATION

The Haskell language is hard to read for readers with no experience with it. Even for readers with a programming background, Haskell language requires extra effort to understand a given assertion.

Figure 13 was generated by using explanations provided by StormBound developers, who show how to interpret the assertions. The "data term" indicates the most inner part of the assertions. The most minute part of the data is constructed by "var Text" and "Const Integer." "Var Text" refers to a variable with the name defined in the text after “Var,” and “Const Integer” refers to a constant with the value of integer. The functions “Add,” “Mul,” “Sub,” “Div,” and “Mod” are used to produce terms from two other terms. They are basic mathematical operations used for addition, subtraction, multiplication, division, and the modular of two values. “Deref” is the dereferencing function, which returns the value stored in the pointed location. “SizeOf” is the function that generates the size of a given variable. “ArrayIx” is the function that stores a sequential collection of elements of the same type. “ArrayIx” function has two parameters: first, showing its name; and second, showing element numbers of the stated array. “Getfield” is the function that searches the public field with the specified name.

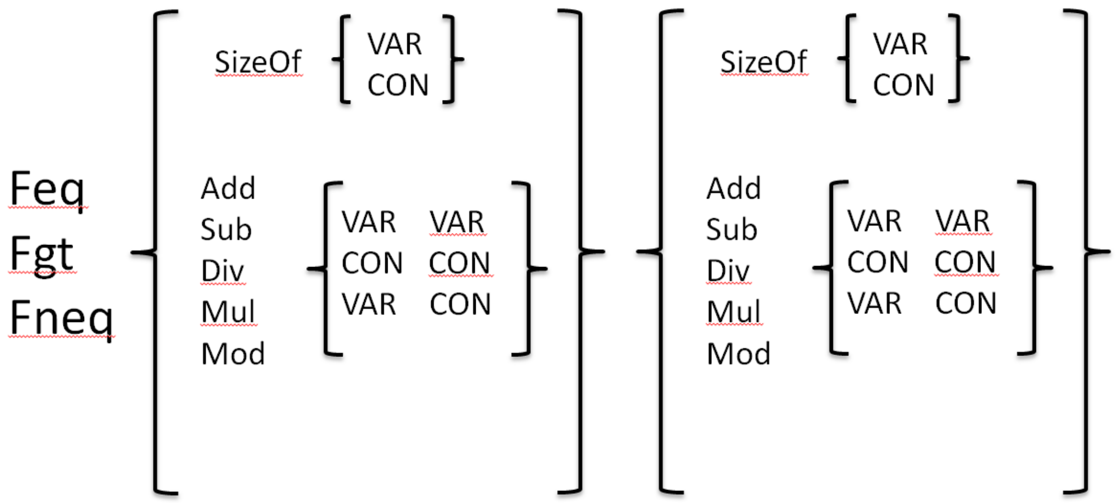
Figure 13. Data Abstraction Table



Third Level		Second Level		First Level	
Output	Input	Output	Input	Output	Input
Assertion	FTrue Ffalse FNot Formula FAnd Formula Formula FOr Formula Formula	Formula	FEq Term Term FGt Term Term FNeq Term Term FValid Term Term Term	Term	Var Text Const Integer Null Error Add Term Term Sub Term Term Mul Term Term Div Term Term Mod Term Term Deref Term SizeOf Term ArrayIx Term Term GetField Term Text

The raw assertion data as represented in Figure 10 was too complex to analyze. That is why we developed an abstraction methodology. We categorized the expression into three levels. Each level expression consists of either the same or lower-level expressions. The first level consists of the operations with terms that are called *variables* and *constants*. *Variables* are the data that have different values throughout the execution of the loop, while the *constant* always has the same value. The output of the first-level operation is another term. Then comes the second level, which takes the output from the first level. Second-level expressions are the most common expressions. They consist of two first-level expressions. Figure 13 shows the organization of a second-level expression. These expressions refer equality, non-equality, or greatness. FEq is the function of equality. It has two term inputs. It compares them, and if they are equal, then it returns true. Otherwise, it returns false. FGt is the function of greater. It indicates greatness between two terms. FNeq is the inverse of FEq. The FValid formula states that a pointer is valid at the first address for the offsets between the second and third terms. The third-level expressions are FTrue, FFalse, FNot, FAnd, and FOr. Third-level expressions have at most two formulas coming from the second level. These formulas represent either "True" or "False." The hierarchical structure of the data is shown in the Figure . Each level produces an output value that can be used in the same level or one level below as an input value. Figure 14 shows an example of second level operations in a loop invariant.

Figure 14. Second-Level Operations Overview



In Figure 15, the upper expression shows a typical StormBound assertion, and the lower expression shows its visual explanation. In the first level, a constant is added into a variable called "level." In the second level, the output of the previous addition is compared with another variable "message_0." In the third level, the output of the previous expression has performed an "or" operation with FTrue. While reading the data, readers should be careful pairing the parenthesis and referring the parenthesis to the right function. The average number of parentheses in the data is 6.5. The total number of parenthesis is 314,730.

Figure 15. Data Translation Example

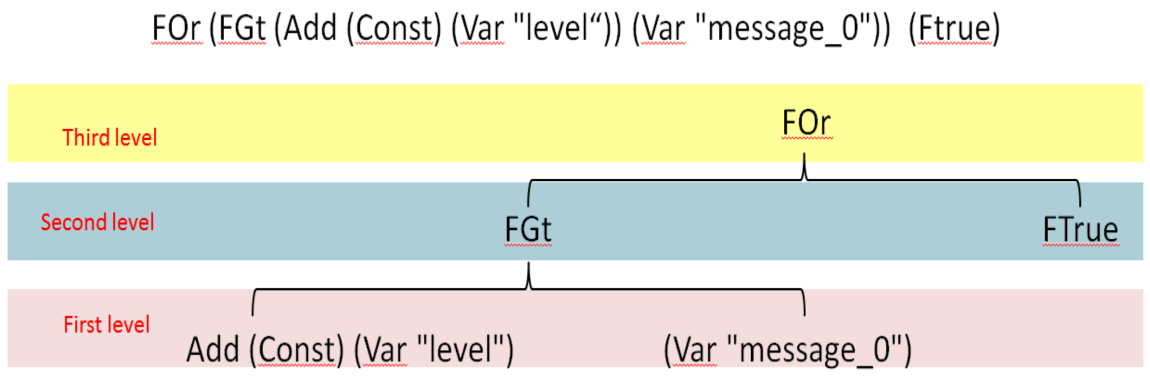
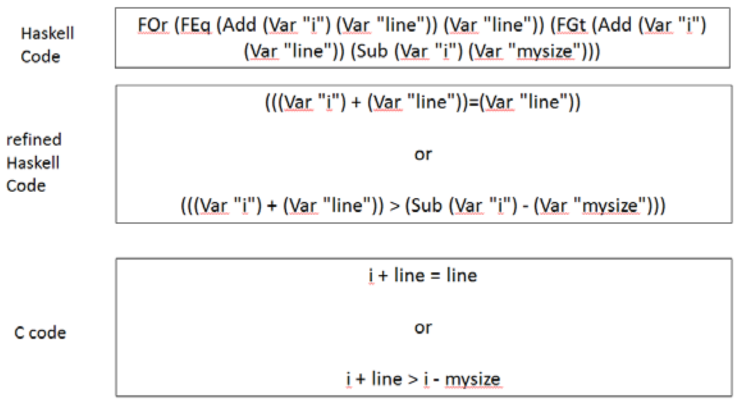

FOr (FGt (Add (Const) (Var "level")) (Var "message_0")) (Ftrue)
Third level
FOr
Second level
FGt
FTrue
First level
Add (Const) (Var "level")
(Var "message_0")

Figure 16 shows a translation of one of the assertions into C language. For the readers with no programming background, assertions basically have two equations: (i+line=line) and (i+line > i-mysize). The assertion combines them with an “or” function. This means that if one of them is true, the assertion is true. Readers find the translated expression easier to understand. However, the example in Figure 16 is one of the short loop invariants inside the data. There are many longer expressions inside the data.

Figure 16. Translation of a Loop Invariant


Haskell Code
FOr (FEq (Add (Var "i") (Var "line")) (Var "line")) (FGt (Add (Var "i") (Var "line")) (Sub (Var "i") (Var "mysize")))
refined Haskell Code
(((Var "i") + (Var "line"))=(Var "line"))
or
(((Var "i") + (Var "line")) > (Sub (Var "i") - (Var "mysize")))
C code
i + line = line
or
i + line > i - mysize

C. DATA MANAGEMENT

Translating one assertion from Haskell language into C language is simple; however, there are 48,244 assertions that need to be analyzed. That burdensome task requires an automation system. We recognized that our data should be translated into a more concise and comprehensible format. Considering the high number of data, such a task should be done by a computer program. For that purpose, we developed a parser, which turns the variants written in Haskell language into more comprehensible format, which makes the manual analysis easier. The codes of parser are shown in the Appendix.A. The parser reads the assertion from a text file named “input” and writes the translated version into a text file named “output.”

Figure 17 shows the main algorithm used in parser. It loads each loop invariant provided into “input” text file as a string variable called assertion. A “for” statement used to read each char inside the assertion variable. It checks if the char value is one of the correspondence letter. In case of the algorithm in Figure 17, it checks for FEq statement. Before running the parser, we changed the all “FEq” statements into “Q” letter to make it easier for the program to match the referred statement. The regular expression library of c programming language can be used to shorten the process. When the parser detects a “Q” letter, it checks the char variable coming after “Q” letter. If that value is a “(“ sign, that indicates that an equality function starts. Then the code calls the pairing function which find the “)” as a pair of a given “(.” Then the code knows that this is an equality function and the numbers of the parenthesis corresponding with the function. The second part of the code changes the “Q” letter with and “=“ sing and replace it after the paired parenthesis.

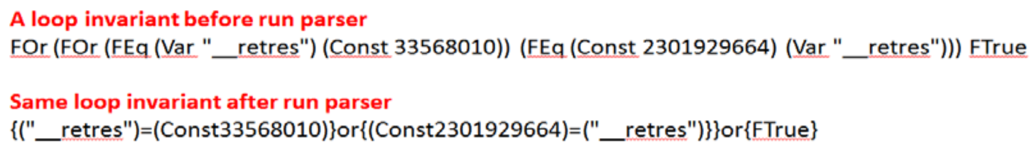
Figure 17. Parser Algorithm

Algorithm 1 Parser Algorithm

```
procedure MyProcedure
    $assertion \leftarrow$ StormBound Loop Invariant
    for each char in $assertion$ do
        $pair \leftarrow FindPair(charNumber + 1, assertion)$
        if $assertion(i) = 'Q'$ then
            replace char with the next one until $pair$
            $assertion(pair+1) \leftarrow '='$
```

As an example, `FGt(“msgset”)(Sub(Const)(SizeOf(“sock”)))` is translated into `msgset>(Const)-SizeOf(“sock”)` by our Parser. The assertions show that a variable named “msfset” is greater than the number produced by subtracting a constant value from the size of the variable named “sock.” The difference between two expressions is so clear that the second one is easier to analyze. Parser was developed with C programming language. It facilitates the data analysis process well. We were able to see the variants. Figure 18 shows an example of loop invariant translated by parser. Clearly, second representation is easy to read for a human expert. The example loop in Figure 18 has three statements connected to each other with OR functions. The second statement in the same figure shows the translation of the first statement performed by parser.

Figure 18. A loop invariant translated by parser



A loop invariant before run parser

```
FOr (FOr (FEq (Var "__retres") (Const 33568010))  (FEq (Const 2301929664)  (Var "__retres"))) FTrue
```

Same loop invariant after run parser

```
{("__retres")=(Const33568010)}or{(Const2301929664)=("__retres")}}or{FTrue}
```

D. SUMMARY

Variants detected by StormBound gamers are stored as expressions in Haskell language. Haskell style is not as clearly understandable as other languages, such as C, C++, or Java. To analyze the data better, gamers' variants should be translated from Haskell language to a more comprehensible format. Parser is a tool developed by C programming language to read the invariants written with Haskell language and write the translated variants into a text file.

THIS PAGE INTENTIONALLY LEFT BLANK

IV. METHODOLOGY AND FINDINGS

This chapter proposes the methodology used in the thesis. Besides using LID gamers, there are two other methods for detecting a loop invariant: expert human detection and machine (computer) detection. Expert solutions are known to be high quality while machine solutions are known to be low quality for the reasons explained in the previous chapter. We propose that machine detectability of a given loop invariant is decent criteria for quality assessment of a solution of a LID gamer.

Therefore, the first section elaborates on why we choose machine detectability. The second section elaborates on how we can determine the machine detectability of a loop invariant. The third section proposes an automated tool to determine machine detectability of multiple loop invariants. We used that tool to examine the 48244 loop invariants detected by StormBound gamers. The fourth section demonstrates our examinations.

A. CRITERIA: MACHINE DETECTABILITY

The CSFV project aims to find a way of using a crowd for formal verification. The realization of that aim depends largely on the satisfactory quality of the solutions (assertions) produced by online gamers. StormBound game requests from players to solve a puzzle the solution of which also produces loop invariants for the related piece of code. Consequently, its overall success primarily depends on the quality of players' invariants. One way of measuring quality is to compare one item to a corresponding item. That does not provide absolute accuracy; however, that type of measuring gives some idea about the quality of the referred item in cases where it is not possible to measure quality in any other way. In the case of LID, a counterpart can be one of the other detection methods. There are two conventional loop invariant detection methods: human experts and automated system.

To make a quality comparison between assertions of StormBound and assertions of human experts is clearly the most accurate and useful way; however, for many reasons, doing so is extremely hard. Firstly, experts have different qualification levels. Similarly, StormBound gamers have different skills. Making a comparison between those two different samples does not provide healthy results. Secondly, making an expert and a StormBound gamer solve the same problem is too hard.

Besides human expertise, some computer programs (automated systems) can detect loop invariants from a given piece of code. Having a limited checklist makes automated systems detect poor quality invariants. Automated systems are blind for detecting any other invariant not stated in the checklist. Using an automated system as a counterpart is a better approach becaues their quality is known to be poor. Because automated systems detect poor quality invariants, the checklist can be used to evaluate any given loop invariant. In this study, we chose the automated systems as the counterpart for our methodology. The relative quality over the automated system of a given assertion detected by a StormBound user is the criteria we used. The invariants detected by an automated system are regarded as having poor quality.

In this study, we defined an invariant that can be detectable by an automated system as machine-detectable. We propose that the machine detectability is good quality criteria for gamers' efforts. The checklist (capability list) of an automated tool can be used to determine whether a given invariant is machine-detectable or not.

B. INVARIANT CHECKLIST OF AN AUTOMATED SYSTEM

The main drawback of an automated LID is its low number of checklist items, which indicates the possible invariants to be detected. These checklists were defined by the developer of the application and can be extended. As a typical example of an automated LID, Daikon can check for 75 different invariants, including being constant ($x = a$), non-zero (x 6= 0), being in a range ($a \leq x \leq b$), linear relationships ($y = ax + b$), ordering ($x \leq y$), functions from a library ($x = fn(y)$), sortedness (x is sorted), and an easily extendable list [20]. In this study, we use the DAIKON checklist for our methodology. The complete Daikon checklist is given in Appendix.B.

There are two ways to check whether a given StormBound assertion is detectable by Daikon. Firstly, we can use the related code to run on DAIKON and compare the result, whether the output of Daikon and StormBound mechanism is same. If they are same, then we know that Daikon can find that invariant; thus, the StormBound game is not necessary to detect that invariant. That is the most accurate method, yet it is hard to accomplish. It requires that each piece of code be run in Daikon and the result recorded.

Secondly, we can use the Daikon checklist to see whether a given assertion is inside that list. If it is inside that list, we can easily affirm that this particular assertion can be detected by Daikon and therefore know that it has poor quality. In this study, we use the second method.

The checklist comparison requires a checking table that was filled with both looking into the Daikon checklist and the assertions detected by StormBound gamers. It is really hard to build such a table. This table needs to have two main columns: functions used in StormBound assertions, and their detectability by Daikon.

The assertions have many subfunctions and need to be divided into small parts for a better diagnosis. Daikon ignores variable names when inferring invariants. In this study, we ignored the name of the variables and regarded them just as "VARIABLES" [24], or "VAR," as a more condensed format. Similarly, all constant values were regarded as "CONSTANT" or "CON," regardless of value.

Secondly, we can shrink the data more by making the following translations. Assuming that a variable added to a constant is another variable, Add(VARIABLE)(CONSTANT) can be regarded as another VARIABLE. The conversion table (see Table 2) shows the possible conversions.

Table 2. Conversion Table

Add(CON)(VAR)	VAR	**Div(CON)(VAR)**	VAR	**ArrayIx(VAR)(CON)**	VAR
Add(VAR)(CON)	VAR	**Div(CON)(CON)**	CON	**GetField(VAR)NAME**	VAR
Add(CON)(CON)	CON	**Div(VAR)(CON)**	VAR	**Add(VAR)(VAR)**	VAR
Sub(CON)(VAR)	VAR	**Mod(CON)(VAR)**	VAR	**Sub(VAR)(VAR)**	VAR
Sub(CON)(CON)	CON	**Mod(CON)(CON)**	CON	**Mul(VAR)(VAR)**	VAR
Sub(VAR)(CON)	VAR	**Mod(VAR)(CON)**	VAR	**Div(VAR)(VAR)**	VAR
Mul(CON)(VAR)	VAR	**Deref(VAR)**	VAR	**Mod(VAR)(VAR)**	VAR
Mul(CON)(CON)	CON	**SizeOf(CON)**	CON		
Mul(VAR)(CON)	VAR	**SizeOf(VAR)**	VAR		

C. AN AUTOMATED TOOL TO CHECK DETECTABILITY

Checking machine detectability of a single invariant is not that hard. However, checking hundreds of them requires automation. For that purpose, we developed the checker, an Excel file with a VBA module. In our model, we assumed that all variables are independent from other variables inside the same invariant and that their names are unimportant. Before running the checker, we deleted all variable names from the invariants.

The checker has four sheets named "main," "assumptions," "detectability," and "results." In the "main" sheet, the user adds the loop invariants to be checked. In the "assumption" sheet, the user defines basic assumptions to be implemented into the invariants. For example, `Add(Variable)(Constant)` is equal to another variable. Implementing these assumptions into a given invariant basically simplifies it. After that simplification operation, the number of variables becomes the minimum. For example, the expression `Add(Add(Variable)(Constant))(Constant)` becomes `(Variable)` after simplification. Our assumptions include the following:

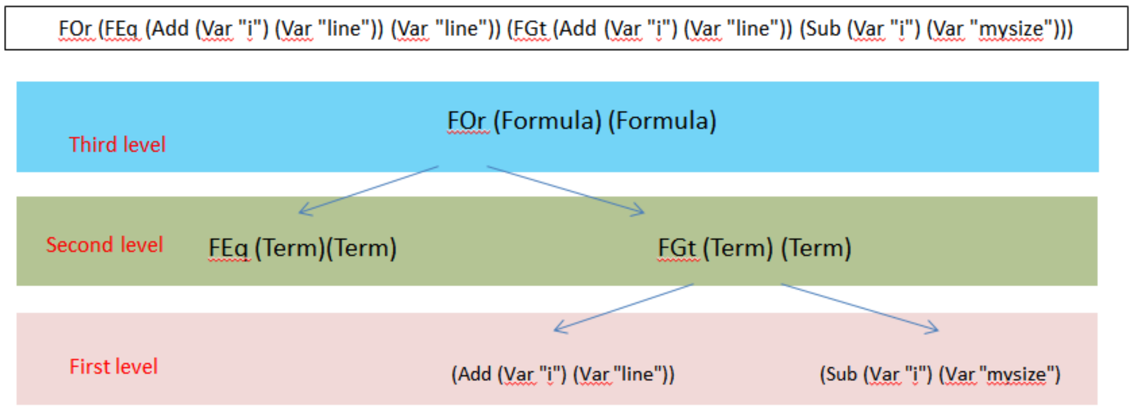
- Output of any first-level operation with two variables is variable.
- Output of any first-level operation with one constant and a variable is variable.
- Output of any first-level operation with two constants is variable.
- Output of any first-level operation with one constant is a constant.
- Output of any first-level operation with one variable is a variable.

Secondly, the checker checks for detectability of the invariant that is simplified in the previous process by using the detectability table in the sheet "detectability." We produced the detectability table by using the Daikon checklist (see Appendix B). Table 3 shows our detectability table. We assumed that the second-level expressions stated in Table 3 are machine detectable. In other words, Daikon can easily detect them.

Table 3. Detectability Table

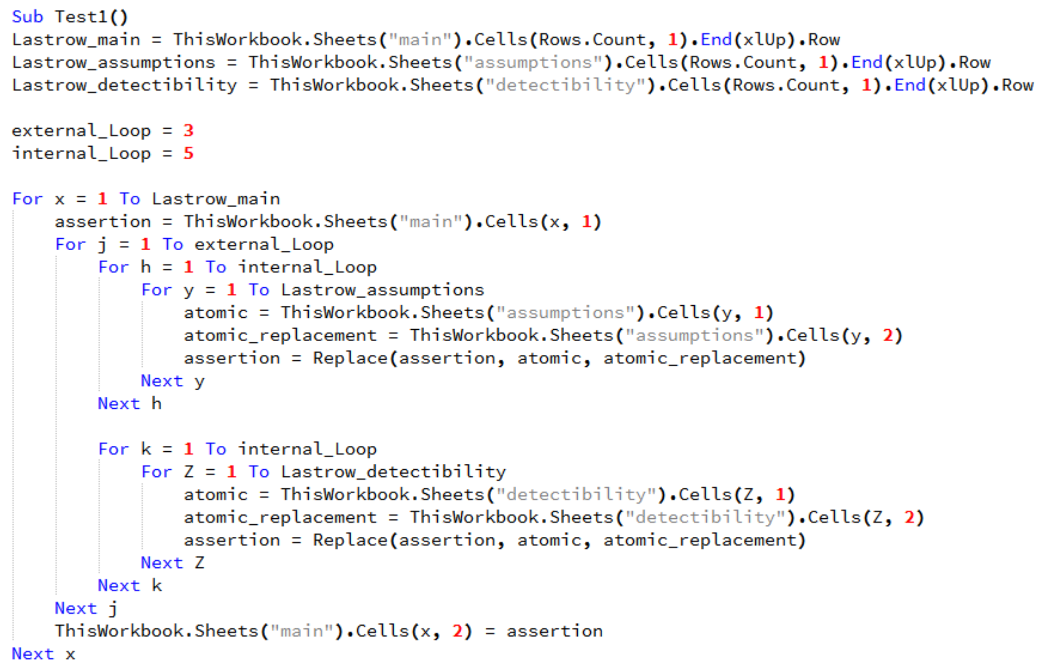
FEq(VAR)(VAR)	FGt(CON)(CON)
FEq(CON)(VAR)	FGt(VAR)Null
FEq(VAR)(CON)	FNeq(VAR)(VAR)
FEq(CON)(CON)	FNeq(CON)(VAR)
FEq(VAR)Null	FNeq(VAR)(CON)
FGt(VAR)(VAR)	FNeq(CON)(CON)
FGt(CON)(VAR)	FNeq(VAR)Null
FGt(VAR)(CON)	

Figure 19. Detectability


FOr (FEq (Add (Var "i") (Var "line")) (Var "line")) (FGt (Add (Var "i") (Var "line")) (Sub (Var "i") (Var "mysize")))
Third level
FOr (Formula) (Formula)
Second level
FEq (Term)(Term)
FGt (Term) (Term)
First level
(Add (Var "i") (Var "line"))
(Sub (Var "i") (Var "mysize")

In Figure 19, the upper figure shows a typical invariant, and the lower figure shows a decomposed form of it in three layers that was explained in the third chapter. The first-level operations consist of many recursive first-level operations. On the other hand, second-level operations are not recursive. The checker has two loops: one for the first level, and one for the second level. The first loop simplifies the first-level expression into either “VARIBLE” or “CONSTANT.” And the second loop checks for detectability. Figure 20 shows the VBA code of the checker.

Figure 20. Excel VBA Codes for Checker



```
Sub Test1()
Lastrow_main = ThisWorkbook.Sheets("main").Cells(Rows.Count, 1).End(xlUp).Row
Lastrow_assumptions = ThisWorkbook.Sheets("assumptions").Cells(Rows.Count, 1).End(xlUp).Row
Lastrow_detectibility = ThisWorkbook.Sheets("detectibility").Cells(Rows.Count, 1).End(xlUp).Row

external_Loop = 3
internal_Loop = 5

For x = 1 To Lastrow_main
    assertion = ThisWorkbook.Sheets("main").Cells(x, 1)
    For j = 1 To external_Loop
        For h = 1 To internal_Loop
            For y = 1 To Lastrow_assumptions
                atomic = ThisWorkbook.Sheets("assumptions").Cells(y, 1)
                atomic_replacement = ThisWorkbook.Sheets("assumptions").Cells(y, 2)
                assertion = Replace(assertion, atomic, atomic_replacement)
            Next y
        Next h

        For k = 1 To internal_Loop
            For Z = 1 To Lastrow_detectibility
                atomic = ThisWorkbook.Sheets("detectibility").Cells(Z, 1)
                atomic_replacement = ThisWorkbook.Sheets("detectibility").Cells(Z, 2)
                assertion = Replace(assertion, atomic, atomic_replacement)
            Next Z
        Next k
    Next j
    ThisWorkbook.Sheets("main").Cells(x, 2) = assertion
Next x
```

We developed checker by using the Microsoft Excel VBA development tool. It can easily be developed by using another language, such as Java, C.

D. FINDINGS

Using assumptions criteria stated in previous section and using the detectability criteria stated in Table 3, we found that 37,718 of the assertions are machine detectable. That number refers to 78% of the whole data. When we deleted the following detectability criteria from the productivity sheet, we observed that the detectability number decreased to 37,130.

Table 4. Weak Assumptions

Add(VAR)(VAR)
Sub(VAR)(VAR)
Mul(VAR)(VAR)
Div(VAR)(VAR)
Mod(VAR)(VAR)

This shows that any change in the assumptions changes the output value. We deleted the value inside Table 4 from detection table, because we believe that these are the weakest assumptions. Deleting those from the assumptions list increases the accuracy. Each assumption requires a more detailed analysis.

E. SUMMARY

The CSFV project needs criteria to measure the quality of gamers' efforts. This study proposes that machine detectability can be used as quality criteria. The checker is a tool that can be used to find out if a particular loop invariant is machine detectable. According to the result from the checker, 78% of the invariants produced by StormBound Players are machine detectable.

THIS PAGE INTENTIONALLY LEFT BLANK

V. CONCLUSIONS AND FUTURE WORK

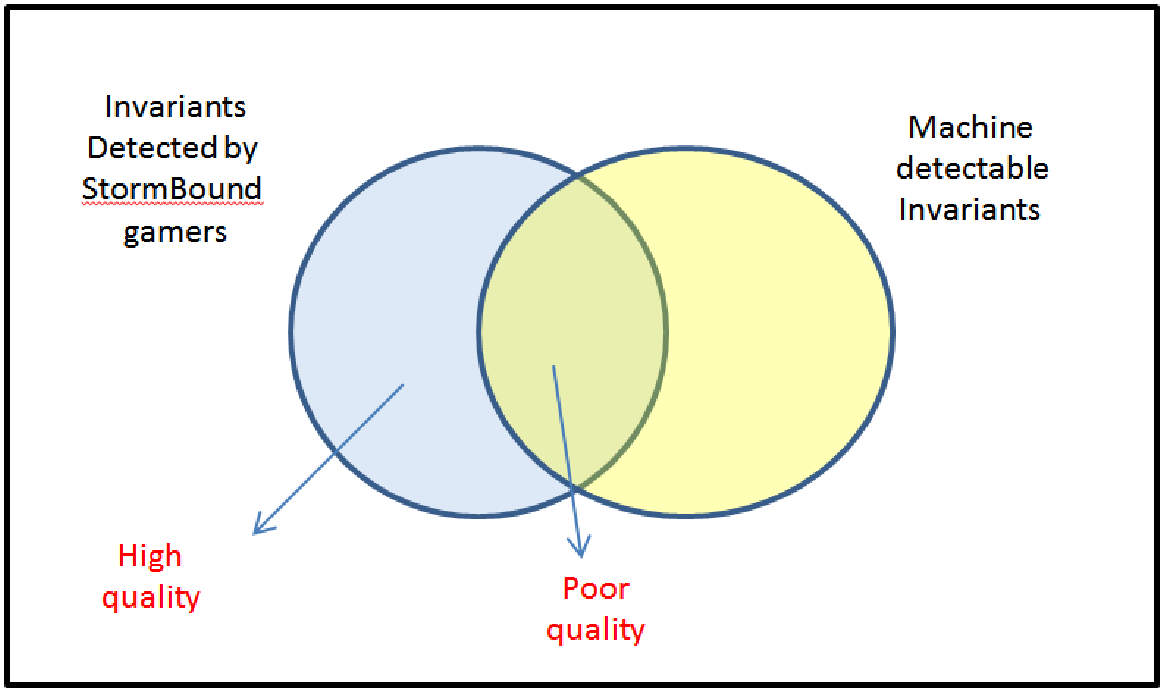
A. SUMMARY AND CONCLUSION

The main goal of this study is to propose criteria for the loop invariants produced by players of StormBound , a CSFV game. This study proposes machine detectability as the quality criteria. Machine detectability refers to whether a given loop invariant that was produced by a CSFV gamer can be detected by an automated loop invariant detector. This study used Daikon as an example automated loop invariant detector. Moreover, this study provided two algorithms: a parser that translates an invariant written in Haskell language into a more understandable format and a checker that determines whether an invariant can be detected by Daikon. Finally, this study states the percentage of machine detectability of invariants detected by gamers.

In this study, we searched a method to measure the quality of the loop invariants produced by StormBound gamers. Firstly, we examined the 48,244 loop invariants collected by the StormBound team and developed a parser to turn them into a more understandable format. Secondly, we proposed that machine detectability can be used as the quality criteria. Thirdly, we examined the invariants for machine detectability. Finally, we developed a script to automate that examination process.

Figure 21 shows the essence of this study. The left circle refers to the loop invariants detected by StormBound players, while the right circle refers to the checklist of automated systems. The area that shows the overlapping parts of both circles indicates the poor quality loop invariants.

Figure 21. Quality Criteria of Loop Invariants


Invariants Detected by StormBound gamers
Machine detectable Invariants
High quality
Poor quality

According to the result from the checker, 78% of the invariants in the dataset are machine detectable and therefore of poor quality. Consequently, 22% of the invariants in the dataset are not machine detectable and therefore are potentially of high quality. As an overall contribution, the steps and criteria used for conducting this work provides a systematic method to measure the quality of loop invariants produced by CSFV gamers, and human analysts in general. This study demonstrates that the stormbound game can potentially be used to attract Internet users to produce loop invariants that would otherwise require human expert effort.

B. LIMITATIONS AND FUTURE WORK

These findings depend on several assumptions, and therefore, the accuracy of the results partially depends on the accuracy of the assumptions. More accurate assumptions require expert level studies in automated loop invariant detection. Secondly, we only used Daikon to represent automated LID, but there are additional tools. Studies should be performed adjusting the assumptions referring to the other tools. Thirdly, it would be beneficial to apply the proposed methodology to study output from other CSFV games in order to properly evaluate the potential of the CSFV concept. Finally, we developed the checker tool with Microsoft Excel and it requires Excel software to run. Studies should be performed to develop a general tool to check the machine detectability of a given loop invariant on other platforms. That tool could increase the functionality of the checker.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX A. PARSER CODES

```
/* Parser for Haskell to C */
//Parser reads the input file line by line and write them in c into output file

#include<stdio.h>
#include <string.h>

int findPair(int first,char assertion[]);

int findPair(int first,char assertion[])
{
        int Parenthesis=0;
        int a;
        if (assertion[first]!='(')
                {
                        return 0;
                }
a=first;

        while( assertion[a] !=  '\0' )
          {
                        if ( assertion[a] == '(' )
                        {
                Parenthesis=Parenthesis+1;
                        }else if (assertion[a]==')')
                        {
                                Parenthesis=Parenthesis-1;
                        }
                        if (Parenthesis==0)
                        {
                                return a;
                        }
                a=a+1;
          }
}

void SpaceRemover(char* statement);

void SpaceRemover(char* statement)
{
  char* first = statement;
  char* second = statement;
  while(*second != 0)
  {
   *first = *second++;
   if(*first != ' ')
     first++;
  }
  *first = 0;
}

int control;
main()
{
        FILE *inputFile;
        inputFile = fopen ("input.txt","r");

        FILE *outputFile;
        outputFile = fopen ("output.txt","w");

        char changeLetter;
        int a,j,pair,pair2,pair3;

        char assertion[2500];
        char temp[2500];

        int nextLine=0;

        while (feof(inputFile) == 0)
        {
                nextLine = nextLine +1;
                puts ("---------------------------");
                fscanf (inputFile, "%s \n", assertion);


int control;
/*in the input.txt file change FTrue into (FTrue) and Null into (Null)*/

//in the input.txt file change (Var" into ("
//in the input.txt file change FOr into V
//in the input.txt file change FNot into C
//in the input.txt file change FEq into Q
//in the input.txt file change FGt into G
//in the input.txt file change FNeq into N
//in the input.txt file change Add into A
//in the input.txt file change Sub into S
//in the input.txt file change Mul into M
//in the input.txt file change Div into D
```

```
//in the input.txt file change FValid into AV
//in the input.txt file change the ArrayIx(Var"name")(Var"name") into
J("name")("name"), this function does just ArrayIx(V...
//in the input.txt file change GetField( into PP(
//in the input.txt file change Deref( into Z(

a=0;
while( assertion[a] !=  '\0' )
  {
                pair= findPair(a+1,assertion);
                if (assertion[a]=='R' & pair!=0)
                {
                        for (j=a; j<=pair; j=j+1)
                        {
                        assertion[j]=assertion[j+1];
                        }
                        assertion[pair]='|';
                        pair3=findPair(pair+1,assertion);
                        assertion[a]='{';
                        assertion[pair-1]='}';
                        assertion[pair+1]='{';
                        assertion[pair3]='}';

                }
                        a=a+1;
}
//FNot
a=0;
while( assertion[a] !=  '\0' )
{
        pair= findPair(a+1,assertion);

        if (assertion[a]=='C' & pair!=0)
                {
                //      printf ("%s" ,"hello");
                        assertion[a+1]='{';
                        assertion[pair]='}';
                        assertion[a]='!';
                }
                        a=a+1;
}
//FEq
a=0;
while( assertion[a] !=  '\0' )
{
```

```
pair= findPair(a+1,assertion);
 if (assertion[a]=='Q'& pair!=0)
                    {
                              for (j=a; j<=pair; j=j+1)
                              {
                              assertion[j]=assertion[j+1];
                              }
                              assertion[pair]='=';
                    }
                              a=a+1;
}
//FGt
a=0;
while( assertion[a] !=  '\0' )
{
pair= findPair(a+1,assertion);
 if (assertion[a]=='G'& pair!=0)
                    {
                              for (j=a; j<=pair; j=j+1)
                              {
                              assertion[j]=assertion[j+1];
                              }
                              assertion[pair]='>';
                     }
                              a=a+1;
}
//FNeq
a=0;
while( assertion[a] !=  '\0' )
{
pair= findPair(a+1,assertion);
if (assertion[a]=='N'& pair!=0)
                    {
                              for (j=a; j<=pair; j=j+1)
                              {
                              assertion[j]=assertion[j+1];
                              }
                              assertion[pair]='!';
                    }
                              a=a+1;
}
//Add
a=0;
while( assertion[a] !=  '\0' )
{
```

```
          pair= findPair(a+1,assertion);
 if (assertion[a]=='A'& pair!=0)
                    {
                              for (j=a; j<=pair; j=j+1)
                              {
                              assertion[j]=assertion[j+1];
                              }
                              assertion[pair]='+';
                    }
                              a=a+1;
}
//Sub
a=0;
while( assertion[a] !=  '\0' )
{
          pair= findPair(a+1,assertion);
          if (assertion[a]=='S'& pair!=0)
                    {
                              for (j=a; j<=pair; j=j+1)
                              {
                              assertion[j]=assertion[j+1];
                              }
                              assertion[pair]='-';
                    }
                              a=a+1;
}
//Mul
a=0;
while( assertion[a] !=  '\0' )
{
pair= findPair(a+1,assertion);
                    if (assertion[a]=='M'& pair!=0)
                    {
                              for (j=a; j<=pair; j=j+1)
                              {
                              assertion[j]=assertion[j+1];
                              }
                              assertion[pair]='x';
                    }
                              a=a+1;
}
//Div
a=0;
while( assertion[a] !=  '\0' )
{
```

```
          pair= findPair(a+1,assertion);

                     if (assertion[a]=='D'& pair!=0)
                    {
                              for (j=a; j<=pair; j=j+1)
                              {
                              assertion[j]=assertion[j+1];
                              }
                              assertion[pair]='/';
                    }
                              a=a+1;
}
//FValid
a=0;
while( assertion[a] !=  '\0' )
{
          pair= findPair(a+1,assertion);
           if (assertion[a]=='V'& pair!=0)
                    {
                              for (j=a; j<=pair; j=j+1)
                              {
                              assertion[j-1]=assertion[j+1];
                              }
                              assertion[pair-1]='f';

                              pair2= findPair(pair+1,assertion);
                              for (j=pair; j<=pair2; j=j+1)
                              {
                              assertion[j]=assertion[j+1];
                              }
                              assertion[pair2]='f';
                    }
                              a=a+1;
}
//ArrayIx
a=0;
while( assertion[a] !=  '\0' )
{
pair= findPair(a+1,assertion);
                    if (assertion[a]=='J'& pair!=0)
                    {
                              pair3=findPair(pair+1,assertion);

                              assertion[a]=' ';
                              assertion[a+1]=' ';
```

```
                                assertion[a+2]=' ';
                                assertion[pair]=' ';
                                assertion[pair-1]=' ';
                                assertion[pair+1]='[';
                                assertion[pair3]=']';

                        }
                                a=a+1;
}
//Getfield
a=0;
while( assertion[a] !=  '\0' )
{
            pair= findPair(a+1,assertion);
                        if (assertion[a]=='P'& pair!=0)

                        {
                                for (j=a; j<=pair; j=j+1)
                                {
                                assertion[j-1]=assertion[j+1];
                                }
                                assertion[pair-1]='-';
                                assertion[pair]='>';
                        }
                                a=a+1;
}
//Deref
a=0;
while( assertion[a] !=  '\0' )
{
            pair= findPair(a+1,assertion);
             if (assertion[a]=='Z' & pair!=0)
                        {
                                assertion[a]=' ';
                                assertion[a+1]='*';
                                assertion[pair]=' ';
                        }
                                a=a+1;
}
//refine {{{
a=0;
while( assertion[a] !=  '\0' )
{
            if (assertion[a]=='{' & assertion[a+1]=='{')
            {
                assertion[a]=' ';
        }
        a=a+1;
}
SpaceRemover(assertion);
fprintf (outputFile, "%s\n" , assertion);

}

fclose(outputFile);
fclose(inputFile);
return 0;
}
```

APPENDIX B. DAIKON CHECKLIST

Daikon checklist consists of the loop invariant types that Daikon can detect. It was drawn from Daikon Invariant Detector User Manual [24].

AndJoiner:This is a special invariant used internally by Daikon to represent an antecedent invariant in an implication where that antecedent consists of two invariants anded together.

CommonFloatSequence:Represents sequences of double values that contain a common subset. Prints as {e1, e2, e3, ...} subset of x[].

CommonSequence:Represents sequences of long values that contain a common subset. Prints as {e1, e2, e3, ...} subset of x[].

CommonStringSequence:Represents string sequences that contain a common subset. Prints as "{s1, s2, s3, ...} subset of x[]".

CompleteOneOfScalar:Tracks every unique value and how many times it occurs.

CompleteOneOfString:Tracks every unique value and how many times it occurs.

DummyInvariant:This is a special invariant used internally by Daikon to represent invariants whose meaning Daikon doesn't understand. The only operation that can be performed on a DummyInvariant is to print it.

EltLowerBound:Represents the invariant that each element of a sequence of long values is greater than or equal to a constant. Prints as x[] elements &gt;= c.

EltLowerBoundFloat:Represents the invariant that each element of a sequence of double values is greater than or equal to a constant. Prints as x[] elements &gt;= c.

EltNonZero:Represents the invariant "x != 0" where x represents all of the elements of a sequence of long. Prints as x[] elements != 0.

EltNonZeroFloat:Represents the invariant "x != 0" where x represents all of the elements of a sequence of double. Prints as x[] elements != 0.

EltOneOf:Represents sequences of long values where the elements of the sequence take on only a few distinct values. Prints as either x[] == c (when there is only one value), or as x[] one of {c1, c2, c3} (when there are multiple values).

EltOneOfFloat:Represents sequences of double values where the elements of the sequence take on only a few distinct values. Prints as either x[] == c (when there is only one value), or as x[] one of {c1, c2, c3} (when there are multiple values).

EltOneOfString:Represents sequences of String values where the elements of the sequence take on only a few distinct values. Prints as either x[] == c (when there is only one value), or as x[] one of {c1, c2, c3} (when there are multiple values).

EltRangeFloat.EqualMinusOne:Internal invariant representing double scalars that are equal to minus one. Used for non-instantiating suppressions. Will never print since OneOf accomplishes the same thing

EltRangeFloat.EqualOne:Internal invariant representing double scalars that are equal to one. Used for non-instantiating suppressions. Will never print since OneOf accomplishes the same thing

EltRangeFloat.EqualZero:Internal invariant representing double scalars that are equal to zero. Used for non-instantiating suppressions. Will never print since OneOf accomplishes the same thing.

EltRangeFloat.GreaterEqual64:Internal invariant representing double scalars that are greater than or equal to 64. Used for non-instantiating suppressions. Will never print since Bound accomplishes the same thing

EltRangeFloat.GreaterEqualZero:Internal invariant representing double scalars that are greater than or equal to 0. Used for non-instantiating suppressions. Will never print since Bound accomplishes the same thing

EltRangeInt.BooleanVal:Internal invariant representing longs whose values are always 0 or 1. Used for non-instantiating suppressions. Will never print since OneOf accomplishes the same thing.

EltRangeInt.Bound0_63:Internal invariant representing longs whose values are between 0 and 63. Used for non-instantiating suppressions. Will never print since Bound accomplishes the same thing.

EltRangeInt.EqualMinusOne:Internal invariant representing long scalars that are equal to minus one. Used for non-instantiating suppressions. Will never print since OneOf accomplishes the same thing

EltRangeInt.EqualOne:Internal invariant representing long scalars that are equal to one. Used for non-instantiating suppressions. Will never print since OneOf accomplishes the same thing

EltRangeInt.EqualZero:Internal invariant representing long scalars that are equal to zero. Used for non-instantiating suppressions. Will never print since OneOf accomplishes the same thing.

EltRangeInt.Even:Invariant representing longs whose values are always even. Used for non-instantiating suppressions. Since this is not covered by the Bound or OneOf invariants it is printed.

EltRangeInt.GreaterEqual64:Internal invariant representing long scalars that are greater than or equal to 64. Used for non-instantiating suppressions. Will never print since Bound accomplishes the same thing

EltRangeInt.GreaterEqualZero:Internal invariant representing long scalars that are greater than or equal to 0. Used for non-instantiating suppressions. Will never print since Bound accomplishes the same thing

EltRangeInt.PowerOfTwo:Invariant representing longs whose values are always a power of 2 (exactly one bit is set). Used for non-instantiating suppressions. Since this is not covered by the Bound or OneOf invariants it is printed.

EltUpperBound:Represents the invariant that each element of a sequence of long values is less than or equal to a constant. Prints as x[] elements &lt;= c.

EltUpperBoundFloat:Represents the invariant that each element of a sequence of double values is less than or equal to a constant. Prints as x[] elements &lt;= c.

EltwiseFloatEqual:Represents equality between adjacent elements (x[i], x[i+1]) of a double sequence. Prints as x[] elements are equal.

EltwiseFloatGreaterEqual:Represents the invariant &ge; between adjacent elements (x[i], x[i+1]) of a double sequence. Prints as x[] sorted by &ge;.

EltwiseFloatGreaterThan:Represents the invariant &gt; between adjacent elements (x[i], x[i+1]) of a double sequence. Prints as x[] sorted by &gt;.

EltwiseFloatLessEqual:Represents the invariant &le; between adjacent elements (x[i], x[i+1]) of a double sequence. Prints as x[] sorted by &le;.

EltwiseFloatLessThan:Represents the invariant &lt; between adjacent elements (x[i], x[i+1]) of a double sequence. Prints as x[] sorted by &lt;.

EltwiseIntEqual:Represents equality between adjacent elements (x[i], x[i+1]) of a long sequence. Prints as x[] elements are equal.

EltwiseIntGreaterEqual:Represents the invariant &ge; between adjacent elements (x[i], x[i+1]) of a long sequence. Prints as x[] sorted by &ge;.

EltwiseIntGreaterThan:Represents the invariant &gt; between adjacent elements (x[i], x[i+1]) of a long sequence. Prints as x[] sorted by &gt;.

EltwiseIntLessEqual:Represents the invariant &le; between adjacent elements (x[i], x[i+1]) of a long sequence. Prints as x[] sorted by &le;.

EltwiseIntLessThan:Represents the invariant &lt; between adjacent elements (x[i], x[i+1]) of a long sequence. Prints as x[] sorted by &lt;.

Equality:Keeps track of sets of variables that are equal. Other invariants are instantiated for only one member of the Equality set, the leader. If variables x, y, and z are members of the Equality set and xis chosen as the leader, then the Equality will internally convert into binary comparison invariants that print as x == y and x == z.

FloatEqual:Represents an invariant of == between two double scalars.

FloatGreaterEqual:Represents an invariant of &ge; between two double scalars.

FloatGreaterThan:Represents an invariant of &gt; between two double scalars.

FloatLessEqual:Represents an invariant of &le; between two double scalars.

FloatLessThan:Represents an invariant of &lt; between two double scalars.

FloatNonEqual:Represents an invariant of != between two double scalars.

FunctionBinary.BitwiseAndLong_{xyz, yxz, zxy}:Represents the invariant x = BitwiseAnd (y, z) over three long scalars. Since the function is symcriteria, only the permutations xyz, yxz, and zxy are checked.

FunctionBinary.BitwiseOrLong_{xyz, yxz, zxy}:Represents the invariant x = BitwiseOr (y, z) over three long scalars. Since the function is symcriteria, only the permutations xyz, yxz, and zxy are checked.

FunctionBinary.BitwiseXorLong_{xyz, yxz, zxy}:Represents the invariant x = BitwiseXor (y, z) over three long scalars. Since the function is symcriteria, only the permutations xyz, yxz, and zxy are checked.

FunctionBinary.DivisionLong_{xyz, xzy, yxz, yzx, zxy, zyx}:Represents the invariant x = Division (y, z) over three long scalars. Since the function is non-symcriteria, all six permutations of the variables are checked.

FunctionBinary.GcdLong_{xyz, yxz, zxy}:Represents the invariant x = Gcd (y, z) over three long scalars. Since the function is symcriteria, only the permutations xyz, yxz, and zxy are checked.

FunctionBinary.LogicalAndLong_{xyz, yxz, zxy}:Represents the invariant x = LogicalAnd (y, z) over three long scalars. For logical operations, Daikon treats 0 as false and all other values as true. Since the function is symcriteria, only the permutations xyz, yxz, and zxy are checked.

FunctionBinary.LogicalOrLong_{xyz, yxz, zxy}:Represents the invariant x = LogicalOr (y, z) over three long scalars. For logical operations, Daikon treats 0 as false and all other values as true. Since the function is symcriteria, only the permutations xyz, yxz, and zxy are checked.

FunctionBinary.LogicalXorLong_{xyz, yxz, zxy}:Represents the invariant x = LogicalXor (y, z) over three long scalars. For logical operations, Daikon treats 0 as false and all other values as true. Since the function is symcriteria, only the permutations xyz, yxz, and zxy are checked.

FunctionBinary.LshiftLong_{xyz, xzy, yxz, yzx, zxy, zyx}:Represents the invariant x = Lshift (y, z) over three long scalars. Since the function is non-symcriteria, all six permutations of the variables are checked.

FunctionBinary.MaximumLong_{xyz, yxz, zxy}:Represents the invariant x = Maximum (y, z) over three long scalars. Since the function is symcriteria, only the permutations xyz, yxz, and zxy are checked.

FunctionBinary.MinimumLong_{xyz, yxz, zxy}:Represents the invariant x = Minimum (y, z) over three long scalars. Since the function is symcriteria, only the permutations xyz, yxz, and zxy are checked.

FunctionBinary.ModLong_{xyz, xzy, yxz, yzx, zxy, zyx}:Represents the invariant x = Mod (y, z) over three long scalars. Since the function is non-symcriteria, all six permutations of the variables are checked.

FunctionBinary.MultiplyLong_{xyz, yxz, zxy}:Represents the invariant x = Multiply (y, z) over three long scalars. Since the function is symcriteria, only the permutations xyz, yxz, and zxy are checked.

FunctionBinary.PowerLong_{xyz, xzy, yxz, yzx, zxy, zyx}:Represents the invariant x = Power (y, z) over three long scalars. Since the function is non-symcriteria, all six permutations of the variables are checked.

FunctionBinary.RshiftSignedLong_{xyz, xzy, yxz, yzx, zxy, zyx}:Represents the invariant x = RshiftSigned (y, z) over three long scalars. Since the function is non-symcriteria, all six permutations of the variables are checked.

FunctionBinary.RshiftUnsignedLong_{xyz, xzy, yxz, yzx, zxy, zyx}:Represents the invariant x = RshiftUnsigned (y, z) over three long scalars. Since the function is non-symcriteria, all six permutations of the variables are checked.

FunctionBinaryFloat.DivisionDouble_{xyz, xzy, yxz, yzx, zxy, zyx}:Represents the invariant x = Division (y, z) over three double scalars. Since the function is non-symcriteria, all six permutations of the variables are checked.

FunctionBinaryFloat.MaximumDouble_{xyz, yxz, zxy}:Represents the invariant x = Maximum (y, z) over three double scalars. Since the function is symcriteria, only the permutations xyz, yxz, and zxy are checked.

FunctionBinaryFloat.MinimumDouble_{xyz, yxz, zxy}:Represents the invariant x = Minimum (y, z) over three double scalars. Since the function is symcriteria, only the permutations xyz, yxz, and zxy are checked.

FunctionBinaryFloat.MultiplyDouble_{xyz, yxz, zxy}:Represents the invariant x = Multiply (y, z) over three double scalars. Since the function is symcriteria, only the permutations xyz, yxz, and zxy are checked.

GuardingImplication:This is a special implication invariant that guards any invariants that are over variables that are sometimes missing. For example, if the invariant a.x = 0 is true, the guarded implication is a != null \rArr; a.x = 0.

Implication:The Implication invariant class is used internally within Daikon to handle invariants that are only true when certain other conditions are also true (splitting).

IntEqual:Represents an invariant of == between two long scalars.

IntGreaterEqual:Represents an invariant of &ge; between two long scalars.

IntGreaterThan:Represents an invariant of &gt; between two long scalars.

IntLessEqual:Represents an invariant of &le; between two long scalars.

IntLessThan:Represents an invariant of &lt; between two long scalars.

IntNonEqual:Represents an invariant of != between two long scalars.

IsPointer:IsPointer is an invariant that heuristically determines whether an integer represents a pointer (a 32-bit memory address). Since both a 32-bit integer and an address have the same representation, sometimes a a pointer can be mistaken for an integer. When this happens, several scalar invariants are computed for integer variables. Most of them would not make any sense for pointers. Determining whether a 32-bit variable is a pointer can thus spare the computation of many irrelevant invariants.

LinearBinary:Represents a Linear invariant between two long scalars x and y, of the form ax + by + c = 0. The constants a, b and c are mutually relatively prime, and the constant a is always positive.

LinearBinaryFloat:Represents a Linear invariant between two double scalars x and y, of the form ax + by + c = 0. The constants a, b and c are mutually relatively prime, and the constant a is always positive.

LinearTernary:Represents a Linear invariant over three long scalars x, y, and z, of the form ax + by + cz + d = 0. The constants a, b, c, and d are mutually relatively prime, and the constant a is always positive.

LinearTernaryFloat:Represents a Linear invariant over three double scalars x, y, and z, of the form ax + by + cz + d = 0. The constants a, b, c, and d are mutually relatively prime, and the constant a is always positive.

LowerBound:Represents the invariant <pre>x &gt;= c</pre>, where c is a constant and x is a long scalar.

LowerBoundFloat:Represents the invariant <pre>x &gt;= c</pre>, where c is a constant and x is a double scalar.

Member:Represents long scalars that are always members of a sequence of long values. Prints as x in y[] where x is a long scalar and y[] is a sequence of long.

MemberFloat:Represents double scalars that are always members of a sequence of double values. Prints as x in y[] where x is a double scalar and y[] is a sequence of double.

MemberString:Represents String scalars that are always members of a sequence of String values. Prints as x in y[] where x is a String scalar and y[] is a sequence of String.

Modulus:Represents the invariant x == r (mod m) where x is a long scalar variable, r is the (constant) remainder, and m is the (constant) modulus.

NoDuplicates:Represents sequences of long that contain no duplicate elements. Prints as x[] contains no duplicates.

NoDuplicatesFloat:Represents sequences of double that contain no duplicate elements. Prints as x[] contains no duplicates.

NonModulus:Represents long scalars that are never equal to r (mod m) where all other numbers in the same range (i.e., all the values that x doesn't take from min(x) to max(x)) are equal to r (mod m). Prints as x != r (mod m), where r is the remainder and m is the modulus.

NonZero:Represents long scalars that are non-zero. Prints as x != 0, or as x != null for pointer types.

NonZeroFloat:Represents double scalars that are non-zero. Prints as x != 0.

NumericFloat.Divides:Represents the divides without remainder invariant between two double scalars. Prints as x % y == 0.

NumericFloat.Square:Represents the square invariant between two double scalars. Prints as x = y**2.

NumericFloat.ZeroTrack:Represents the zero tracks invariant between two double scalars; that is, when x is zero, y is also zero. Prints as x = 0 &rArr; y = 0.

NumericInt.BitwiseAndZero:Represents the BitwiseAnd == 0 invariant between two long scalars; that is, x and y have no bits in common. Prints as x &amp; y == 0.

NumericInt.BitwiseComplement:Represents the bitwise complement invariant between two long scalars. Prints as x = ~y.

NumericInt.BitwiseSubset:Represents the bitwise subset invariant between two long scalars; that is, the bits of y are a subset of the bits of x. Prints as x = y | x.

NumericInt.Divides:Represents the divides without remainder invariant between two long scalars. Prints as x % y == 0.

NumericInt.ShiftZero:Represents the ShiftZero invariant between two long scalars; that is, x right-shifted by y is always zero. Prints as x &gt;&gt; y = 0.

NumericInt.Square:Represents the square invariant between two long scalars. Prints as x = y**2.

NumericInt.ZeroTrack:Represents the zero tracks invariant between two long scalars; that is, when x is zero, y is also zero. Prints as x = 0 &rArr; y = 0.

OneOfFloat:Represents double variables that take on only a few distinct values. Prints as either x == c (when there is only one value) or as x one of {c1, c2, c3} (when there are multiple values).

OneOfFloatSequence:Represents double[] variables that take on only a few distinct values. Prints as either x == c (when there is only one value) or as x one of {c1, c2, c3} (when there are multiple values).

OneOfScalar:Represents long scalars that take on only a few distinct values. Prints as either x == c (when there is only one value), x one of {c1, c2, c3} (when there are multiple values), or x has only one value (when x is a hashcode (pointer) - this is because the numerical value of the hashcode (pointer) is uninteresting).

OneOfSequence:Represents long[] variables that take on only a few distinct values. Prints as either x == c (when there is only one value) or as x one of {c1, c2, c3} (when there are multiple values).

OneOfString:Represents String variables that take on only a few distinct values. Prints as either x == c (when there is only one value) or as x one of {c1, c2, c3} (when there are multiple values).

OneOfStringSequence:Represents String[] variables that take on only a few distinct values. Prints as either x == c (when there is only one value) or as x one of {c1, c2, c3} (when there are multiple values).

PairwiseFloatEqual:Represents an invariant between corresponding elements of two sequences of double values. The length of the sequences must match for the invariant to hold. A comparison is made over each(x[i], y[i]) pair. Thus, x[0] is compared to y[0], x[1] to y[1], and so forth. Prints as x[] == y[].

PairwiseFloatGreaterEqual:Represents an invariant between corresponding elements of two sequences of double values. The length of the sequences must match for the invariant to hold. A comparison is made over each(x[i], y[i]) pair. Thus, x[0] is compared to y[0], x[1] to y[1], and so forth. Prints as x[] &ge; y[].

PairwiseFloatGreaterThan:Represents an invariant between corresponding elements of two sequences of double values. The length of the sequences must match for the invariant to hold. A comparison is made over each(x[i], y[i]) pair. Thus, x[0] is compared to y[0], x[1] to y[1], and so forth. Prints as x[] &gt; y[].

PairwiseFloatLessEqual:Represents an invariant between corresponding elements of two sequences of double values. The length of the sequences must match for the invariant to hold. A comparison is made over each(x[i], y[i]) pair. Thus, x[0] is compared to y[0], x[1] to y[1], and so forth. Prints as x[] &le; y[].

PairwiseFloatLessThan:Represents an invariant between corresponding elements of two sequences of double values. The length of the sequences must match for the invariant to hold. A comparison is made over each(x[i], y[i]) pair. Thus, x[0] is compared to y[0], x[1] to y[1], and so forth. Prints as x[] &lt; y[].

PairwiseIntEqual:Represents an invariant between corresponding elements of two sequences of long values. The length of the sequences must match for the invariant to hold. A comparison is made over each(x[i], y[i]) pair. Thus, x[0] is compared to y[0], x[1] to y[1], and so forth. Prints as x[] == y[].

PairwiseIntGreaterEqual:Represents an invariant between corresponding elements of two sequences of long values. The length of the sequences must match for the invariant to hold. A comparison is made over each(x[i], y[i]) pair. Thus, x[0] is compared to y[0], x[1] to y[1], and so forth. Prints as x[] &ge; y[].

PairwiseIntGreaterThan:Represents an invariant between corresponding elements of two sequences of long values. The length of the sequences must match for the invariant to hold. A comparison is made over each(x[i], y[i]) pair. Thus, x[0] is compared to y[0], x[1] to y[1], and so forth. Prints as x[] &gt; y[].

PairwiseIntLessEqual:Represents an invariant between corresponding elements of two sequences of long values. The length of the sequences must match for the invariant to hold. A comparison is made over each(x[i], y[i]) pair. Thus, x[0] is compared to y[0], x[1] to y[1], and so forth. Prints as x[] &le; y[].

PairwiseIntLessThan:Represents an invariant between corresponding elements of two sequences of long values. The length of the sequences must match for the invariant to hold. A comparison is made over each(x[i], y[i]) pair. Thus, x[0] is compared to y[0], x[1] to y[1], and so forth. Prints as x[] &lt; y[].

PairwiseLinearBinary:Represents a linear invariant (i.e., y = ax + b) between the corresponding elements of two sequences of long values. Each (x[i], y[i]) pair is examined. Thus, x[0] is compared to y[0], x[1]to y[1] and so forth. Prints as y[] = a * x[] + b.

PairwiseLinearBinaryFloat:Represents a linear invariant (i.e., y = ax + b) between the corresponding elements of two sequences of double values. Each (x[i], y[i]) pair is examined. Thus, x[0] is compared to y[0],x[1] to y[1] and so forth. Prints as y[] = a * x[] + b.

PairwiseNumericFloat.Divides:Represents the divides without remainder invariant between corresponding elements of two sequences of double. Prints as x[] % y[] == 0.

PairwiseNumericFloat.Square:Represents the square invariant between corresponding elements of two sequences of double. Prints as x[] = y[]**2.

PairwiseNumericFloat.ZeroTrack:Represents the zero tracks invariant between corresponding elements of two sequences of double; that is, when x[] is zero, y[] is also zero. Prints as x[] = 0 &rArr; y[] = 0.

PairwiseNumericInt.BitwiseAndZero:Represents the BitwiseAnd == 0 invariant between corresponding elements of two sequences of long; that is, x[] and y[] have no bits in common. Prints as x[] &amp; y[] == 0.

PairwiseNumericInt.BitwiseComplement:Represents the bitwise complement invariant between corresponding elements of two sequences of long. Prints as x[] = ~y[].

PairwiseNumericInt.BitwiseSubset:Represents the bitwise subset invariant between corresponding elements of two sequences of long; that is, the bits of y[] are a subset of the bits of x[]. Prints as x[] = y[] | x[].

PairwiseNumericInt.Divides:Represents the divides without remainder invariant between corresponding elements of two sequences of long. Prints as x[] % y[] == 0.

PairwiseNumericInt.ShiftZero:Represents the ShiftZero invariant between corresponding elements of two sequences of long; that is, x[] right-shifted by y[] is always zero. Prints as x[] &gt;&gt; y[] = 0.

PairwiseNumericInt.Square:Represents the square invariant between corresponding elements of two sequences of long. Prints as x[] = y[]**2.

PairwiseNumericInt.ZeroTrack:Represents the zero tracks invariant between corresponding elements of two sequences of long; that is, when x[] is zero, y[] is also zero. Prints as x[] = 0 &rArr; y[] = 0.

PairwiseString.SubString:Represents the substring invariant between corresponding elements of two sequences of String. Prints as x[] is a substring of y[].

PairwiseStringEqual:Represents an invariant between corresponding elements of two sequences of String values. The length of the sequences must match for the invariant to hold. A comparison is made over each(x[i], y[i]) pair. Thus, x[0] is compared to y[0], x[1] to y[1], and so forth. Prints as x[] == y[].

PairwiseStringGreaterEqual:Represents an invariant between corresponding elements of two sequences of String values. The length of the sequences must match for the invariant to hold. A comparison is made over each(x[i], y[i]) pair. Thus, x[0] is compared to y[0], x[1] to y[1], and so forth. Prints as x[] &ge; y[].

PairwiseStringGreaterThan:Represents an invariant between corresponding elements of two sequences of String values. The length of the sequences must match for the invariant to hold. A comparison is made over each(x[i], y[i]) pair. Thus, x[0] is compared to y[0], x[1] to y[1], and so forth. Prints as x[] &gt; y[].

PairwiseStringLessEqual:Represents an invariant between corresponding elements of two sequences of String values. The length of the sequences must match for the invariant to hold. A comparison is made over each(x[i], y[i]) pair. Thus, x[0] is compared to y[0], x[1] to y[1], and so forth. Prints as x[] &le; y[].

PairwiseStringLessThan:Represents an invariant between corresponding elements of two sequences of String values. The length of the sequences must match for the invariant to hold. A comparison is made over each(x[i], y[i]) pair. Thus, x[0] is compared to y[0], x[1] to y[1], and so forth. Prints as x[] &lt; y[].

Positive:Represents the invariant x &gt; 0 where x is a long scalar. This exists only as an example for the purposes of the manual. It isn't actually used (it is replaced by the more general invariant LowerBound).

PrintableString:Represents a string that contains only printable ascii characters (values 32 through 126 plus 9 (tab)

RangeFloat.EqualMinusOne:Internal invariant representing double scalars that are equal to minus one. Used for non-instantiating suppressions. Will never print since OneOf accomplishes the same thing

RangeFloat.EqualOne:Internal invariant representing double scalars that are equal to one. Used for non-instantiating suppressions. Will never print since OneOf accomplishes the same thing

RangeFloat.EqualZero:Internal invariant representing double scalars that are equal to zero. Used for non-instantiating suppressions. Will never print since OneOf accomplishes the same thing.

RangeFloat.GreaterEqual64:Internal invariant representing double scalars that are greater than or equal to 64. Used for non-instantiating suppressions. Will never print since Bound accomplishes the same thing

RangeFloat.GreaterEqualZero:Internal invariant representing double scalars that are greater than or equal to 0. Used for non-instantiating suppressions. Will never print since Bound accomplishes the same thing

RangeInt.BooleanVal:Internal invariant representing longs whose values are always 0 or 1. Used for non-instantiating suppressions. Will never print since OneOf accomplishes the same thing.

RangeInt.Bound0_63:Internal invariant representing longs whose values are between 0 and 63. Used for non-instantiating suppressions. Will never print since Bound accomplishes the same thing.

RangeInt.EqualMinusOne:Internal invariant representing long scalars that are equal to minus one. Used for non-instantiating suppressions. Will never print since OneOf accomplishes the same thing

RangeInt.EqualOne:Internal invariant representing long scalars that are equal to one. Used for non-instantiating suppressions. Will never print since OneOf accomplishes the same thing

RangeInt.EqualZero:Internal invariant representing long scalars that are equal to zero. Used for non-instantiating suppressions. Will never print since OneOf accomplishes the same thing.

RangeInt.Even:Invariant representing longs whose values are always even. Used for non-instantiating suppressions. Since this is not covered by the Bound or OneOf invariants it is printed.

RangeInt.GreaterEqual64:Internal invariant representing long scalars that are greater than or equal to 64. Used for non-instantiating suppressions. Will never print since Bound accomplishes the same thing

RangeInt.GreaterEqualZero:Internal invariant representing long scalars that are greater than or equal to 0. Used for non-instantiating suppressions. Will never print since Bound accomplishes the same thing

RangeInt.PowerOfTwo:Invariant representing longs whose values are always a power of 2 (exactly one bit is set). Used for non-instantiating suppressions. Since this is not covered by the Bound or OneOf invariants it is printed.

Reverse:Represents two sequences of long where one is in the reverse order of the other. Prints as x[] is the reverse of y[].

ReverseFloat:Represents two sequences of double where one is in the reverse order of the other. Prints as x[] is the reverse of y[].

SeqFloatEqual:Represents an invariant between a double scalar and a a sequence of double values. Prints as x[] elements y where x is a double sequence and y is a double scalar.

SeqFloatGreaterEqual:Represents an invariant between a double scalar and a a sequence of double values. Prints as x[] elements &ge; y where x is a double sequence and y is a double scalar.

SeqFloatGreaterThan:Represents an invariant between a double scalar and a a sequence of double values. Prints as x[] elements &gt; y where x is a double sequence and y is a double scalar.

SeqFloatLessEqual:Represents an invariant between a double scalar and a a sequence of double values. Prints as x[] elements &le; y where x is a double sequence and y is a double scalar.

SeqFloatLessThan:Represents an invariant between a double scalar and a a sequence of double values. Prints as x[] elements &lt; y where x is a double sequence and y is a double scalar.

SeqIndexFloatEqual:Represents an invariant over sequences of double values between the index of an element of the sequence and the element itself. Prints as x[i] == i.

SeqIndexFloatGreaterEqual:Represents an invariant over sequences of double values between the index of an element of the sequence and the element itself. Prints as x[i] &ge; i.

SeqIndexFloatGreaterThan:Represents an invariant over sequences of double values between the index of an element of the sequence and the element itself. Prints as x[i] &gt; i.

SeqIndexFloatLessEqual:Represents an invariant over sequences of double values between the index of an element of the sequence and the element itself. Prints as x[i] &le; i.

SeqIndexFloatLessThan:Represents an invariant over sequences of double values between the index of an element of the sequence and the element itself. Prints as x[i] &lt; i.

SeqIndexFloatNonEqual:Represents an invariant over sequences of double values between the index of an element of the sequence and the element itself. Prints as x[i] != i.

SeqIndexIntEqual:Represents an invariant over sequences of long values between the index of an element of the sequence and the element itself. Prints as x[i] == i.

SeqIndexIntGreaterEqual:Represents an invariant over sequences of long values between the index of an element of the sequence and the element itself. Prints as x[i] &ge; i.

SeqIndexIntGreaterThan:Represents an invariant over sequences of long values between the index of an element of the sequence and the element itself. Prints as x[i] &gt; i.

SeqIndexIntLessEqual:Represents an invariant over sequences of long values between the index of an element of the sequence and the element itself. Prints as x[i] &le; i.

SeqIndexIntLessThan:Represents an invariant over sequences of long values between the index of an element of the sequence and the element itself. Prints as x[i] &lt; i.

SeqIndexIntNonEqual:Represents an invariant over sequences of long values between the index of an element of the sequence and the element itself. Prints as x[i] != i.

SeqIntEqual:Represents an invariant between a long scalar and a a sequence of long values. Prints as x[] elements == y where x is a long sequence and y is a long scalar.

SeqIntGreaterEqual:Represents an invariant between a long scalar and a a sequence of long values. Prints as x[] elements &ge; y where x is a long sequence and y is a long scalar.

SeqIntGreaterThan:Represents an invariant between a long scalar and a a sequence of long values. Prints as x[] elements &gt; y where x is a long sequence and y is a long scalar.

SeqIntLessEqual:Represents an invariant between a long scalar and a a sequence of long values. Prints as x[] elements &le; y where x is a long sequence and y is a long scalar.

SeqIntLessThan:Represents an invariant between a long scalar and a a sequence of long values. Prints as x[] elements &lt; y where x is a long sequence and y is a long scalar.

SeqSeqFloatEqual:Represents invariants between two sequences of double values. If order matters for each variable (which it does by default), then the sequences are compared lexically. Prints as x[] == y[] lexically.

SeqSeqFloatGreaterEqual:Represents invariants between two sequences of double values. If order matters for each variable (which it does by default), then the sequences are compared lexically. Prints as x[] &ge; y[] lexically.

SeqSeqFloatGreaterThan:Represents invariants between two sequences of double values. If order matters for each variable (which it does by default), then the sequences are compared lexically. Prints as x[] &gt; y[] lexically.

SeqSeqFloatLessEqual:Represents invariants between two sequences of double values. If order matters for each variable (which it does by default), then the sequences are compared lexically. Prints as x[] &le; y[] lexically.

SeqSeqFloatLessThan:Represents invariants between two sequences of double values. If order matters for each variable (which it does by default), then the sequences are compared lexically. Prints as x[] &lt; y[] lexically.

SeqSeqIntEqual:Represents invariants between two sequences of long values. If order matters for each variable (which it does by default), then the sequences are compared lexically. Prints as x[] == y[] lexically.

SeqSeqIntGreaterEqual:Represents invariants between two sequences of long values. If order matters for each variable (which it does by default), then the sequences are compared lexically. Prints as x[] &ge; y[] lexically.

SeqSeqIntGreaterThan:Represents invariants between two sequences of long values. If order matters for each variable (which it does by default), then the sequences are compared lexically. Prints as x[] &gt; y[] lexically.

SeqSeqIntLessEqual:Represents invariants between two sequences of long values. If order matters for each variable (which it does by default), then the sequences are compared lexically. Prints as x[] &le; y[] lexically.

SeqSeqIntLessThan:Represents invariants between two sequences of long values. If order matters for each variable (which it does by default), then the sequences are compared lexically. Prints as x[] &lt; y[] lexically.

SeqSeqStringEqual:Represents invariants between two sequences of String values. If order matters for each variable (which it does by default), then the sequences are compared lexically. Prints as x[] == y[] lexically.

SeqSeqStringGreaterEqual:Represents invariants between two sequences of String values. If order matters for each variable (which it does by default), then the sequences are compared lexically. Prints as x[] &ge; y[] lexically.

SeqSeqStringGreaterThan:Represents invariants between two sequences of String values. If order matters for each variable (which it does by default), then the sequences are compared lexically. Prints as x[] &gt; y[] lexically.

SeqSeqStringLessEqual:Represents invariants between two sequences of String values. If order matters for each variable (which it does by default), then the sequences are compared lexically. Prints as x[] &le; y[] lexically.

SeqSeqStringLessThan:Represents invariants between two sequences of String values. If order matters for each variable (which it does by default), then the sequences are compared lexically. Prints as x[] &lt; y[] lexically.

StdString.SubString:Represents the substring invariant between two String scalars. Prints as x is a substring of y.

StringEqual:Represents an invariant of == between two String scalars.

StringGreaterEqual:Represents an invariant of &ge; between two String scalars.

StringGreaterThan:Represents an invariant of &gt; between two String scalars.

StringLessEqual:Represents an invariant of &le; between two String scalars.

StringLessThan:Represents an invariant of &lt; between two String scalars.

StringNonEqual:Represents an invariant of != between two String scalars.

SubSequence:Represents two sequences of long values where one sequence is a subsequence of the other. Prints as x[] is a subsequence of y[].

SubSequenceFloat:Represents two sequences of double values where one sequence is a subsequence of the other. Prints as x[] is a subsequence of y[].

SubSet:Represents two sequences of long values where one of the sequences is a subset of the other; that is each element of one sequence appears in the other. Prints as either x[] is a subset of y[]or as x[] is a superset of y[].

SubSetFloat:Represents two sequences of double values where one of the sequences is a subset of the other; that is each element of one sequence appears in the other. Prints as either x[] is a subset of y[] or as x[] is a superset of y[].

SuperSequence:Represents two sequences of long values where one sequence is a subsequence of the other. Prints as x[] is a subsequence of y[].

SuperSequenceFloat:Represents two sequences of double values where one sequence is a subsequence of the other. Prints as x[] is a subsequence of y[].

SuperSet:Represents two sequences of long values where one of the sequences is a subset of the other; that is each element of one sequence appears in the other. Prints as either x[] is a subset of y[]or as x[] is a superset of y[].

SuperSetFloat:Represents two sequences of double values where one of the sequences is a subset of the other; that is each element of one sequence appears in the other. Prints as either x[] is a subset of y[] or as x[] is a superset of y[].

UpperBound:Represents the invariant <pre>x &lt;= c</pre>, where c is a constant and x is a long scalar.

UpperBoundFloat:Represents the invariant <pre>x &lt;= c</pre>, where c is a constant and x is a double scalar.

LIST OF REFERENCES

[1] H. Logas *et al.*, "Software Verification Games: Designing Xylem, The Code of Plants," 2014.

[2] W. Dietl *et al.*, "Verification games: Making verification fun," in Proceedings of the 14th Workshop on Formal Techniques for Java-Like Programs, Beijing, China, 2012, pp. 42–49.

[3] M Hsieh. (n.d.). Crowd Sourced Formal Verification (CSFV). [Online]. Available: http://www.darpa.mil/program/crowd-sourced-formal-verification.

[4] Z. Durumeric *et al.*,"The matter of heartbleed," in *Proceedings of the 2014 Conference on Internet Measurement Conference*, Vancouver, BC, Canada, 2014, pp. 475–488.

[5] J. M. Wing, "A symbiotic relationship between formal methods and security," in *Computer Security, Dependability and Assurance: From Needs to Solutions*, 1998. Proceedings, Washington, DC, 1998, pp. 26–38.

[6] Y. Moy, E. Ledinot, H. Delseny, V. Wiels and B. Monate, "Testing or Formal Verification: DO-178C Alternatives and Industrial Experience," *Software*, IEEE, vol. 30, pp. 50–57, 2013.

[7] A. J. Ko, B. Dosono and N. Duriseti, "Thirty years of software problems in the news," in *Proceedings of the 7th International Workshop on Cooperative and Human Aspects of Software Engineering*, Hyderabad, India, 2014, pp. 32–39.

[8] H. Shahriar and M. Zulkernine, "Mitigating Program Security Vulnerabilities: Approaches and Challenges," *ACM Comput.Surv*., vol. 44, pp. 11:1-11:46, jun, 2012.

[9] B. P. Miller, L. Fredriksen and B. So, "An Empirical Study of the Reliability of UNIX Utilities," Commun ACM, vol. 33, pp. 32–44, dec, 1990.

[10] Common vulnerabilities and exposures. (2015, Aug. 25). [Online]. Available: http://cve.mitre.org/.

[11] H. Shahriar, H. M. Haddad and I. Vaidya, "Buffer Overflow Patching for C and C++ Programs: Rule-based Approach," SIGAPP Appl.Comput.Rev., vol. 13, pp. 8–19, jun, 2013.

[12] N. H. Pham, T. T. Nguyen, H. A. Nguyen and T. N. Nguyen, "Detection of recurring software vulnerabilities," in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering, Antwerp,* Belgium, 2010, pp. 447–456.

[13] US-CERT bulletins. (n.d.). [Online]. Available: http://www.us-cert.gov/

[14] C. Smith and G. Francia III, “Security fuzzing toolset,” in Proceedings of the 50th Annual Southeast Regional Conference, New York, , 2012, pp. 329–330.

[15] A. Takanen, “Fuzzing: The past, the present and the future,” in *Actes Du 7{\`e}Me Symposium Sur La s{\’E}Curit{\’E} Des Technologies De L’Information Et Des Communications* (SSTIC), Codenomicon Ltd, 2009, pp. 202–212.

[16] K. Y. Sim, F. Kuo and R. Merkel, “Fuzzing the out-of-memory killer on embedded linux: An adaptive random approach,” in *Proceedings of the 2011 ACM Symposium on Applied Computing*, New York, 2011, pp. 387–392.

[17] A. Sanghavi, “What is formal verification?” *EE Times Asia*, 21 May 2010. 2010.

[18] O. Ponsini, H. Collavizza, C. Fedele, C. Michel and M. Rueher, “Automatic verification of loop invariants,” in *Software Maintenance (ICSM), 2010 IEEE International Conference*, Timisoara, 2010, pp. 1–5.

[19] M. Kim and A. Petersen, “An Evaluation of Daikon: A Dynamic Invariant Detector,”. [Online]. Available : https://www.cs.cmu.edu/~aldrich/courses/654-sp05/homework/example-tool-eval.pdf

[20] M. D. Ernst *et al.,*”The Daikon System for Dynamic Detection of Likely Invariants,” *Sci.Comput.Program.*, vol. 69, pp. 35–45, dec, 2007.

[21] K. Deibel. (2002). “On the Automatic Detection of Loop Invariants.” [Online]. http://courses.cs.washington.edu/courses/cse503/02wi/papers/deibel.pdf

[22] C. Gladisch, “Verification-based test case generation for full feasible branch coverage,” in *Software Engineering and Formal Methods, 2008. SEFM’08. Sixth IEEE International Conference*, 2008, pp. 159–168.

[23] Jicheng Fu, F. B. Bastani and I-Ling Yen, “Automated discovery of loop invariants for high-assurance programs synthesized using AI planning techniques,” in *High Assurance Systems Engineering Symposium, 2008. HASE 2008. 11th IEEE*, Washington, DC, 2008, pp. 333–342.

[24] Daikon. (August 4, 2015). The Daikon Invariant Detector User Manual [Daikon Invariant Detector User Manual]. Available: http://plse.cs.washington.edu/daikon/download/doc/daikon.html.

[25] D. C. Brabham, “Crowdsourcing as a Model for Problem Solving -- An Introduction and Cases,” *Convergence*, vol. 14, pp. 75, 2008.

[26] C. Rashtchian, P. Young, M. Hodosh and J. Hockenmaier, “Collecting image annotations using amazon’s mechanical turk,” in *Proceedings of the NAACL HLT 2010 Workshop on Creating Speech and Language Data with Amazon’s Mechanical Turk*, Los Angeles, California, 2010, pp. 139–147.

[27] F. Khatib *et al.*,”Algorithm discovery by protein folding game players,” *Proc. Natl. Acad. Sci.*, vol. 108, pp. 18949-18953, Nov 22, 2011.

[28] F. Khatib *et al.*,”Crystal structure of a monomeric retroviral protease solved by protein folding game players,” *Nature Structural & Molecular Biology*, vol. 18, pp. 1175–1177, 09/2011, 2011.

[29] What is Frama-C. (n.d.) Frama-C Software Analyzers. Available: http://frama-c.com/what_is.html.

[30] S. P. Jones, Ed., Haskell, *98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003.

THIS PAGE INTENTIONALLY LEFT BLANK

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California